The FlexSCADA Binary Encrypted Sync Protocol

Preliminary Draft

The protocol is based on an HTTP server and client model with the FlexSCADA device taking the client roll and the server taking the HTTP server roll and is used to efficiently and securely exchange data readings from the device to a 3rd party server.

Customers have successfully made implementations to this standard in Python, PHP and C++. FlexSCADA can on request provide an example written in C++ that may help the developers in their implementation.

This is a preliminary manual intended for someone with a strong background in computer and science and a comprehensive knowledge of C/C++, specifically with bit structures and the JSON data format.

For a more simplified implementation consider polling the /api/metrics endpoint on the device for a JSON array of all the measurements on the device.

See manual section 4.1 for documentation on this endpoint
https://flexscada.com/manuals/flexs-q5/
For best practice always utilize the HTTPS web server on the device along with signed certificates to ensure end to end encryption.

Please Note: This documentation refers to the Cloud Sync protocol available on the device. The Devices onboard web-server also has an API that can be used to read measurements and set its configuration. If direct IP based device access is available and data usage is not a concern it may be a more practical solution for integration. Some of the devices local web server API is documented in the manual linked above, some endpoints you will have to just use a web browser and web inspector to evaluate the requests made as you utilize functions of the on-board web server.
You can also contact us for assistance with specific areas.

If your company utilizes this protocol we recommend you send an email to info@flexscada.com with your company details so we can inform you in advance of any changes made to this protocol. Generally any changes would maintain backwards compatibility and we are not planning any changes at this time but we still strongly recommend letting us know so we can keep you informed.

Intro

All communication is initiated by the device sending measurements to the server. Below are some examples of the payload content exchanged with the security layer removed for simplicity.

Initial Communication Example:

Device sends measurements to server (meta-data contains UID, Config Version, Device Time, FW, etc)

Since in our example this is the initial communication sequence we assume that the server does not have a local copy of the config or the config version is older than the version existing on the device. So we simply respond to the http request with a payload containing a command instructing the device to upload a copy of it’s configuration to us. The device responds by uploading it’s config to the server.

Server Reply (HTTP Payload):

```
{
“Cmd”:”getcfg”
}
```

HTTP Status Code: 409 (Tells the device that the measurements were not processed, they will be re-sent by the device after the server has a valid device config and can process the measurements)

This command is received by the device and prompts the device to make a post request back to the server at the following url.
<Server URL>/Q5/cfg/<uid>
The post payload contains the current device configuration which is stored in the JSON format.

Subsequent Device to Server Communication Example:

Device sends measurements to server (meta-data contains UID, Config Version, Device Time, FW, etc)

Server processes received measurements based on the devices config file which it now has (More information on this below)

If the measurements were processed correctly they should be acknowledged by sending a HTTP Status code 200 which will instruct the Q5 that it can flush it's queued measurements

Server could respond with a plain code 200 and no body or no response. Or respond with several commands: (HTTP Payload Examples)

Set device time:

```
{
"t":<unix epoch>
}
```

Set device config:

```
{
"cfg":{<device config>}
}
```

Execute list of commands:

```
{
"commands":[
```

Set Relay State

```
{
"cmd":"set_relay",
"ch":<channel #>,
"state":<State (1 or 0)>,
"Id":<auto incrementing command ID, id of last command acknowledged is contained in
device to server request meta-data>
```

},

Pulse Relay (5S Cycle)
{
“cmd”:”pulse_relay”,
“ch”:<channel #>,
“Id”:<auto incrementing command id>
},

Set Custom Feed
{
“cmd”:”set_feed”,
“feed”:<feed_id>,
“value”:<value>,
“Id”:<auto incrementing command id>
},

Reset Relay Fusing
{
“cmd”:”reset_relay”,
“ch”:<channel #>,
“Id”:<auto incrementing command id>
}

]
}

Transport and Security Layer

All communication between the device and the server are based on the HTTP standard with the server acting as an HTTP server and the device acting as an HTTP client.

All payload data is encrypted by 256 bit AES-CBC with a pre-shared key and SHA256 hashing for data integrity.

Currently there are three distinct transmission types that are utilized

- Queued Measurement Uploads **(Device to Server)**
    - Sent at cloud sync interval, most common transmission type
    - All Binary Format
    - Always posted to <Server URL>/Q5/m
- Configuration Uploads **(Device to Server)**
    - Sent whenever the device configuration is updated or the server requests the device upload its current configuration via the "getcfg" command
    - Binary Header
    - Plain-Text JSON payload after decryption
    - Always posted to <Server URL>/Q5/cfg/<Device UID>
- Server to Device Commands **(Server to Device)**
    - **Always sent as a reply to Queued Measurement Uploads**
    - Used to issue device commands to change relays states etc.
    - Used to push new configurations to the device
    - Plain-Text JSON payload after decryption

The following types of transmissions are outlined in more detail throughout the next pages.

Queued Measurement Format

```
typedef struct
{
   uint32_t uid;
   AES_HEADER aes_header; // AES cryptography header
   uint32_t flags; // Used to convey error messages
   uint32_t fw_version; // Device Firmware Version
   uint32_t cfg_version; // Device Configuration Version
   uint32_t measurementCount; // How many queued measurements
   uint32_t measurementSize; // Size of each queued measurement
   uint32_t epoch; // Current Device time
   uint8_t lastCmdIdAck; // ID of last ack'd command
   char payload[]; // payload of queued measurements
} MeasurementPacketHeader;
```

The payload is the queued measurements recorded by the device and consists of a stacked array of type MeasurementHeader. The number of items in this array is contained in the `measurementCount` field in the MeasurementPacketHeader

```
typedef struct
{
    uint32_t timestamp;
    char payload[];
} MeasurementHeader;
```

The timestamp is a unix epoch
The payload for each measurement contains the list of readings for each metric logged on the Q5. (e.g. CH1, CH2, CH3 values, etc)
Since no specific structure is sent in order to conserve data usage the configuration must be held by the server in order to know how to parse the measurements out of this payload, i.e. to know which bytes correspond to CH1 etc.

More information on parsing the measurements in **“Parsing Binary Measurements”** section.

Configuration Upload Format

```
typedef struct
{
    AES_HEADER aes_header; // AES cryptography header
    char payload[]; // payload of configuration (Plain Text JSON after decryption)
} ConfigPacketHeader;
```

Security Layer
As previously mentioned all of the data exchanged is encrypted by AES256 and hashed for integrity by SHA256 with random cipher block chaining (CBC) as an additional security measure

Decrypting received packets

In the MeasurementPacketHeader and ConfigPacketHeader you will notice the field with type AES_HEADER.
This field contains the necessary data to decode the data coming from the device as long as the password to the device is already known.

```
typedef struct AES_HEADER {
    uint32_t payloadLength;
    uint8_t hash[16]; // hash is actually 32 bytes, last 16bytes sets the IV for CBC
    uint8_t aes_iv[16];
} AES_HEADER;
```

The Initialization Vector (IV) is used for the AES-CBC decoding and is generated by the hashing function.

After the payload is decoded the hash should be confirmed, this is done by first saving the received hash to a temporary buffer and then setting the hash field to the 256 bit AES key of the device (does overflow in the AES_IV field) and then computing a SHA256 hash on the entire packet starting at the AES_HEADER field

```
error_t error = sha256Compute((char*)aes_header, aes_header->payloadLength
+ sizeof(AES_HEADER), result_hash);
```

This hash should be compared with the received hash to verify data integrity

Obtaining the pre-shared AES key of the device

The AES key of the device can be found in HEX format by logging into the device and opening the JSON config editor. It can also be generated from the device passphrase with the following steps

Append ‘FlexsQ5!’ to the end of the passphrase and hash using SHA256

I.e. deviceAesKey[32] = sha256(password + “FlexsQ5!);

Server to Device Command Response Format

Commands can be transported to the device as replies to measurement uploads, this removes the need for port forwarding and other routing issues that are often encountered on networks.

Commands sent to the device are always of the following format, the payload syntax and example commands are documented in the **Initial Communication & Subsequent Device to Server Communication Examples**

```
typedef struct
{
   AES_HEADER aes_header; // AES cryptography header
   char payload[]; // payload of command (Plain Text JSON after decryption)
} CommandPacketHeader;
```

Encrypting transmitted packets

All of the plaintext json commands are preceded with a field of type AES_HEADER as seen above in the CommandPacketHeader structure.

```
typedef struct AES_HEADER {
   uint32_t payloadLength;
   uint8_t hash[16]; // hash is actually 32 bytes, last 16bytes sets the IV for CBC
   uint8_t aes_iv[16];
} AES_HEADER;
```

The first step to sending data is to set the payloadLength of the aes_header field to the length of the payload text in bytes.

The payload length must be in increments of 16 bytes for the AES encryption, the payload should be padded with null characters or spaces to an even 16 byte boundaries.

The second step is to set the hash field to the devices pre-shared key, the aes key is 32 bytes and the hash field is only 16 so we let the key overflow into the aes_iv field.

For more information on obtaining the aes key for the device refer to section titled **Obtaining the pre-shared AES key of the device** for more information,

The third step is to calculate the AES hash on the entire payload and to place the result into the hash field (Again, overflowing into the aes_iv field)

```
error_t error = sha256Compute((char*)aes_header, aes_header->payloadLength +
sizeof(AES_HEADER), aes_header->hash);
```

The fourth step is to encrypt the payload using AES256-CBC using the IV from the aes_header

```
 memcpy(modifiable_iv,aes_header->aes_iv,16);
 error_t err = AES-CBC-Encrypt(modifiable_iv, CommandPacketHeader->payload,
CommandPacketHeader->payload,  aes_header->payloadLength);
```

Parsing Binary Measurements

The payload is the queued measurements and consists of a stacked array of type MeasurementHeader. The number of items in this array is contained in the `measurementCount` field in the MeasurementPacketHeader

```
typedef struct
{
    uint32_t timestamp;
    char payload[];
} MeasurementHeader;
```

Each measurement contains a timestamp followed by a packed buffer containing each of the metrics selected to be logged on the Q5 device.

To fully understand the contents of this buffer we start by parsing the JSON config file pulling measurements from the buffer as we advance through it in the following order
We strongly recommend you view a device config and follow each step below to gain a better understanding of the materials shared.

1. **Relays**
    a. Loop through each relay channel in the config file
        i. Loop through each metric in the "logging" array in the channel object
            1. state, fuse, hvd, lvd are *Discrete* type metrics and occupy 1 bit of data which corresponds to a logic true or false
            2. amperage,power are float32 type values
                a. **Note, Each entry in a `logging` array represents a metric that is stored in the measurement payload, different entries occupy different sizes. I.e. true or false (discrete) readings use 1 bit, Float32 type metrics use 32bits. Each time you iterate to a new metric you should parse those bits from the buffer and advance to the next bits. Since we pack the metrics tightly without padding it will be necessary to utilize bit shifting to place the extracted results into variables where their values can be represented correctly.**
2. **Analog Inputs**
    a. Loop through each input channel in the config file
        i. Loop through each metric in the "logging" array in the channel object
            1. All metrics are type float32 (32 bits) except metrics labeled "state" which are discrete (1 bit)
3. **Temperature Sensors**
    a. Loop through each item in the `ds18b20` array in the config file
        i. Loop through each metric in the "logging" array in the sensor object

            1. All metrics are type float32 (32 bits) and can include inst,avg,min and max

4. **Power Metrics**
    a. Loop through each item in the `power_metrics` array in the config file
        i. Loop through each metric in the “logging” array in the sensor object
            1. All metrics are type float32 (32 bits) and can include a wide number of metrics for various power measurements
5. **Custom Feeds**
    a. Loop through each item in the `mfeeds` array in the config file
        i. Loop through each metric in the “logging” array in the sensor object
            1. Two types of metrics exist for manual feeds, `state` and `value` which correspond to a discrete type reading and a float32 type reading respectively

6. **Plugins**
    a. Measurements for enabled plugins will follow the above measurements if enabled, since these measurements are very implementation dependent they are not covered in this manual but would generally follow the same syntax as the above metrics. Any bits more than the measurement size extending beyond what was parsed shall be ignored by the server parsing the data. This is often the case as the measurement size is always padded to 8 bits to ensure correct alignment

After parsing the measurement the parser would advance to the next queued measurement and process it in the same manner.