Extract from thread initiated by Matthijs on the Victron forum archive:
https://communityarchive.victronenergy.com/questions/187303/victron-bluetooth-advertising-protocol.html
His post includes the "extra-manufacturer-data-2022-12-14.pdf" as an attachment (via this download hyperlink):
https://communityarchive.victronenergy.com/storage/attachments/extra-manufacturer-data-2022-12-14.pdf

Following are posts in the thread comments much further down the page,
the first by Jake Baldwin (19 May 2023) and
the other by Chris Jackson (3 Nov 2023)

==========

Jake Baldwin (19 May 2023)

---

Jake provided the following additional context/supplement to the original "Extra manufacturing Data" document "as it leaves quite a bit to be desired".

Here is a practical example with some pitfalls I encountered:

1) Advertisement Key Retrieval: As far as I could tell there is no way to get the advertisement keys from the Windows App, iPhone app, or Android app. I was able to get the key from my friends Mac and I've read you can also get it from Linux but haven't tried it myself.

2) Raspberry Pi: I wasn't able to obtain the key from a Raspberry Pi either. The current Raspberry Pi OS BT driver won't pair or connect to the Victron Controller so you can't get the keys. The Victron maintained Raspberry Pi image, VenusOS, doesn't allow connecting to their devices via BT, only serial, which validates the issues I was having; even Victron couldn't figure it out!

However, the BT driver in Raspberry Pi OS can listen for the the BLE advertisement messages which is great if you've already obtained the keys.

3) BLE Advertisement Data: This was particularly unclear from the provided documentation. Here are 6 BLE advertisement data packets I got from one of my SmartSolar MPPT 100/50 charge controllers:

```
10 02 57 a0 01 00 02 20 b0 08 09 0f 73 44 a4 e6 6f 7d 00 24
10 02 57 a0 01 10 02 20 e4 c5 b2 08 2a b8 8d 83 33 7e 4f cc
10 02 57 a0 01 20 02 20 17 a5 b2 d8 02 bf ca 8c 4a c3 ad 39
10 02 57 a0 01 30 02 20 d2 de 94 01 9e 56 6e 03 d7 23 f1 b2
10 02 57 a0 01 40 02 20 58 bd 6b 2c ea ab 78 6d 7a 5b f1 4f
10 02 57 a0 01 50 02 20 b9 40 bc c1 66 e5 38 97 66 dc 4a d8
```

Byte [0] is the Manufacturer Data Record type and is always 0x10.
Byte [1] and [2] are the model id. In my case the 0x02 0x57 means I have the MPPT 100/50 (I forget where I found this info but it's always 2 bytes and it's not really needed for decryption)
Byte [3] is the "read out type" which was always 0xA0 in my case but i didn't use this byte at all

The first 4 bytes aren't mentioned in the provided documentation so it was difficult to figure out where the "extra data" started. Now we get into the bytes documented:
Byte [4] is the record type. In my case it was always 0x01 because I have a "Solar Charger"

Byte [5] and [6] are the Nonce/Data Counter used for decryption (more on this later)
Byte [7] should match the first byte of your devices encryption key. In my case this was 0x20.

The rest of the bytes are the encrypted data of which there are 12 bytes for my Victron device.

4) Decryption: If you have the Encryption Key and the Nonce/Data Counter (from bytes [5] and [6]) in the BLE advertisement data you can decrypt the packet. The description in the provided documentation is not the best. I had to use reverse engineer the Python library referenced above in this thread because I'm using C#. Here is the Python library function in question:

https://github.com/keshavdv/victron-ble/blob/841f5601800a70b35df910526f174427a5e39c12/victron_ble/devices/base.py#L388

Now there are a few issues here.
1) The Python library pads the data bytes to get a full AES block of 16 bytes, and the documentation refers to using "1 AES operation" to decrypt the data if less than or equal to 16-bytes. This suggests a block cipher algorithm. However, the AES-CTR is not a block cipher algorithm, it's a stream cipher algorithm. This means no block extension should be necessary because the data is decrypted byte by byte, that's what a stream cipher means. In my case 12-bytes in gives me 12-bytes out.


2) The "Nonce/Data Counter" referred to in the documentation is often referred to as the "Salt" or the "Initialization Vector" in encryption lexicon which made a google search difficult


3) Now this was the worst part! The "Nonce/Data Counter" (bytes [5] and [6]) are not only transmitted in the advertisement data LSB first (little endian), but must also be used in the decryption as a little endian value! The provided documentation doesn't mention this fact at all!!! I only figured this out from looking at the referenced github Python library. Now what does this actually mean in practice if you aren't using Python? Because if you run the python code in a debugger it sets up a new counter object in little endian format but it puts it in an obscure object where you don't have access to the underlying byte array. Take this BLE advertisement packet as an example:

10 02 57 a0 01 40 02 20 58 bd 6b 2c ea ab 78 6d 7a 5b f1 4f

Byte [5] and [6] are the Nonce/Data Counter, in this case it's 0x40 0x02. Put this Nonce/Counter/Salt/Initialization Vector a 16 byte array in little endian format it should look like this:

{ 0x40 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 }

4) AES counter mode isn't supported in C# but you can see this stack overflow answer for an algorithm that works: https://stackoverflow.com/a/51188472. Just use the counter as described previously.

==========

Chris Jackson (3 Nov 2023)
---
I've been following this thread with interest for some time, and finally, it seems all the pieces have come together. So I'm embarking on developing an Arduino/ESP32-based solution to access realtime Advertisement data from my two 100/30 MPPT controllers.

Apologies for the length of what follows....

I have started with the "BLE-Scan" example in the Arduino IDE, and running it in my ESP32 recognises my two MPPT controllers:

Example output from BLE-Scan:
Advertised Device: Name: MPPT Port, Address: ff:01:6f:13:e7:3b, manufacturer data: e10210024aa001870b9009f6f5e5cecf272b95552e6b, rssi: -93
Advertised Device: Name: MPPT Stbd, Address: cc:18:68:b6:4f:d4, manufacturer data: e102100276a001bd073651a0cc59bb4d94d966b29049, rssi: -95

Some successive "manufacturer data" records for the Stbd device (from advertisedDevice.getManufacturerData()), eg:

MPPT Stbd: e102100276a001ca6e363ae821389810e0ee0315568c
MPPT Stbd: e102100276a001d16e367a16705dbfc2fd72b1cfedde
MPPT Stbd: e102100276a001ec6e365a7d0892e2b974afe10cb98c
MPPT Stbd: e102100276a001f76e36e2f52ad7bd23912f5f4c0d4e
MPPT Stbd: e102100276a001fa6e365bd3fb3c6cd3a343d225048d
MPPT Stbd: e102100276a001326f367a286066dc43595b827d15b2
MPPT Stbd: e102100276a0013b6f365e9d850ad6964623559abfb8
MPPT Stbd: e102100276a001416f36963d7871dd70f1e5c320733d
MPPT Stbd: e102100276a001476f36f5543d0966d0a36647bafa54
MPPT Stbd: e102100276a0014f6f368f5ab68b399ed92e0c66dd2f
MPPT Stbd: e102100276a0015d6f3633615c8132fa6c7eb2e6bc86

My hope was that the "manufacturer data" above would contain the data I want in encrypted form, plus a 7-byte preamble as described by @Jake Baldwin above. But I can't reconcile my preamble with what @Jake Baldwin describes, ie:

Byte [0], e1, expecting 10 (record type)
Byte [1-2], 0210, feasible (device ID; mine is 100/30, 100/50 is 0257)
Byte [3], 02, expecting a0 (readout type)
Byte [4], 76, expecting 01 (record type, Solar Charger is 01)
Byte [5-6], a001, expecting an incrementing value (nonce)
Byte [7], variable, expecting 33 (the first byte of my Encryption Key "36b...")

I've also had a look at the Payload, which similarly doesn't seem to correspond with expectations.
Here are the public properties of the class I'm using:

```
class BLEAdvertisedDevice {
public:
BLEAdvertisedDevice();
BLEAddress getAddress();
uint16_t getAppearance();
std::string getManufacturerData();
std::string getName();
int getRSSI();
BLEScan* getScan();
std::string getServiceData();
std::string getServiceData(int i);
BLEUUID getServiceDataUUID();
```

```
BLEUUID getServiceDataUUID(int i);
BLEUUID getServiceUUID();
BLEUUID getServiceUUID(int i);
int getServiceDataCount();
int getServiceDataUUIDCount();
int getServiceUUIDCount();
int8_t getTXPower();
uint8_t* getPayload();
size_t getPayloadLength();
esp_ble_addr_type_t getAddressType();
void setAddressType(esp_ble_addr_type_t type);
bool isAdvertisingService(BLEUUID uuid);
bool haveAppearance();
bool haveManufacturerData();
bool haveName();
bool haveRSSI();
bool haveServiceData();
bool haveServiceUUID();
bool haveTXPower();
std::string toString();

I have a suspicion that what I'm seeing labelled as "manufacturer data" is not the same as "Extra Manufacturer Data", which is apparently what I need. Can anyone please shed some light on what might be going on? TIA!!

Second post by Chris Jackson (same day):
---------------------------------------

"...I now see that if I discard the first two bytes (e1 02), the rest of the data matches my expectations perfectly. So I'm now moving on to the decryption task..."

--- end ---
```