ESP MODEM

This component is used to communicate with modems in the command mode (using AT commands), as well as the data mode (over PPPoS protocol). The modem device is modeled with a DCE (Data Communication Equipment) object, which is composed of:

- DTE (Data Terminal Equipment), which abstracts the terminal (currently only UART implemented).
- PPP Netif representing a network interface communicating with the DTE using PPP protocol.
- Module abstracting the specific device model and its commands.

```
+-----+
| DTE |--+
+-----+  |   +-------+
         +-->|   DCE |
+-------+    |       |o--- set_mode(command/data)
| Module|--->|       |
+-------+    |       |o--- send_commands
          +->|       |
+------+  |  +-------+
| PPP  |--+
| netif|------------------> network events
+------+
```

Modem components

DCE

This is the basic operational unit of the esp_modem component, abstracting a specific module in software, which is basically configured by

- the I/O communication media (UART), defined by the DTE configuration
- the specific command library supported by the device model, defined with the module type
- network interface configuration (PPPoS config in lwip)

After the object is created, the application interaction with the DCE is in

- issuing specific commands to the modem
- switching between data and command mode

DTE

Is an abstraction of the physical interface connected to the modem. Current implementation supports only UART

PPP netif

Is used to attach the specific network interface to a network communication protocol used by the modem. Currently implementation supports only PPPoS protocol.

Module

Abstraction of the specific modem device. Currently the component supports SIM800, BG96, SIM7600.

Use cases

Users interact with the esp-modem using the DCE's interface, to basically

- Switch between command and data mode to connect to the internet via cellular network.
- Send various commands to the device (e.g. send SMS)

The applications typically register handlers for network events to receive notification on the network availability and IP address changes.

Common use cases of the esp-modem are also listed as the examples:

- `examples/pppos_client` simple client which reads some module properties and switches to the data mode to connect to a public mqtt broker.
- `examples/modem_console` is an example to exercise all possible module commands in a console application.
- `examples/ap_to_pppos` this example focuses on the network connectivity of the esp-modem and provides a WiFi AP that forwards packets (and uses NAT) to and from the PPPoS connection.

Working modes

Modem devices could work in multiple different modes, esp-modem library uses these states to describe them: - Standard modes:

  - Command mode – This mode is used for sending AT commands
  - Data or PPP mode – This mode is used for data communication (to create PPPoS tunnel between the device and the library)

- **Multiplexing modes:**

    - CMUX mode – This mode creates two virtual channels and uses one for sending AT commands and the other one for data communication.
    - DUAL mode – This mode uses two physical channels the same way as CMUX. This mode is supported only by certain devices, usually with USB interface.
- Manual CMUX modes – These modes are designed for applications to have better control over CMUX mode transitions. It allows setting up the virtual channels, switching between channels, transitioning between data and command modes for each channel separately, and exiting the CMUX.

Switching between common modes

The diagram below depicts allowed transitions between the most common modes

```
+---------+    +---------+
| COMMAND |<-->|   DATA  |
+---------+    +---------+
     ^
     |
     v
+-------+
| CMUX  |
+-------+
```

Note that it is possible to switch from any mode to the “UNDEF” mode and vice-versa.

Switching between manual modes

The diagram below depicts allowed transitions between manual CMUX modes

```
                       +------------------------------------
                       |                                   |
+----------+    +-------------+    +------------+    +----------+
|          |<-->| MANUAL_DATA |<-->| MANUAL_CMD |<-->| COMMAND  |
| CMUX     |    +-------------+    +------------+    |  (CMUX   |
| MANUAL   |           |                             |  MANUAL  |
|          |    +-------------+                      |  EXIT)   |
|          |<-->| MANUAL_SWAP |<-------------------->|          |
+----------+    +-------------+                      +----------+
     |                                                     |
     +-----------------------------------------------------+
```

Note that transitioning between “MANUAL_DATA” and “MANUAL_CMD” switches the secondary terminal (dedicated to PPP session) and could be used for recovering data communication if PPP session gets dropped.

Extensibility

CMUX

Implements virtual terminals which allow users to also issue commands in the data mode; after creating two virtual terminals, designating one of them solely to data mode, and another one solely to command mode.

DTE

Currently, we support only UART (and USB as a preview feature), but modern modules support other communication interfaces, such as USB, SPI.

Other devices

Adding a new device is a must-have requirement for the esp-modem component. Different modules support different commands, or some commands might have a different implementation. Adding a new device means to provide a new implementation as a class derived from `GenericModule`, where we could add new commands or modify the existing ones. If you have to support a custom device with C-API, please refer to the example `examples/pppos_client` and enable `ESP_MODEM_ADD_CUSTOM_MODULE`. For advanced use-case, mainly with C++ API and/or usage of esp_modem's Factory class, please read <advanced_api>.

Configuration

Modem abstraction is configurable both compile-time and run-time.

Component Kconfig

Compile-time configuration is provided using menuconfig. Please check the description for the CMUX mode configuration options.

Runtime configuration

Is defined using standard configuration structures for `DTE` and `DCE` objects separately. Please find documentation of

- :cpp:class: `esp_modem_dte_config_t`
- :cpp:class: `esp_modem_dce_config_t`

Known issues

There are certain typical issues you might experience when working with esp_modem. This section elaborates on these problems and offers workarounds.

When using UART terminal, you might find running OTA over https quite unstable, exhibiting frequent UART buffer overflows. The reason for that is that ESP32's UART is interrupt driven, so while performing computational extensive tasks with TLS session, we need to timely interrupt to process incoming data. Here'are few suggestions to mitigate the impact: * Move UART ISR to IRAM * Increase internal UART rx buffer size * Increase UART terminal task priority * Use UART flow control If none of the above helps, you can check the test esp_modem/test/target_ota, which performs OTA in two steps – first read the data, then pass the data to mbedTLS. See the test `README.md` for more details.

Potential issues when using CMUX mode and these devices:

1) Some modems (e.g. A76xx serries) use 2 bytes CMUX payload, which might cause buffer overflow when trying to defragment the payload. It's recommended to disable `ESP_MODEM_CMUX_DEFRAGMENT_PAYLOAD`, which will fix the issue, but may occasional cause reception of AT command replies in fragments.

2. Some devices (such as SIM7000) do not support CMUX mode at all.

3) Device A7670 does no not correctly exit CMUX mode. You can apply this patch to adapt the exit sequence https://github.com/espressif/esp-protocols/commit/28de34571012d36f2e87708955dcd435ee5eab70

```
diff --git a/components/esp_modem/src/esp_modem_cmux.cpp
b/components/esp_modem/src/esp_modem_cmux.cpp
index 0c480f8..4418c3d 100644
--- a/components/esp_modem/src/esp_modem_cmux.cpp
+++ b/components/esp_modem/src/esp_modem_cmux.cpp
@@ -206,6 +206,15 @@ bool CMux::on_header(CMuxFrame &frame)
}
size_t payload_offset = std::min(frame.len, 4 - frame_header_offset);
memcpy(frame_header + frame_header_offset, frame.ptr, payload_offset);
+    if (frame_header[1] == 0xEF) {
+        dlci = 0;
+        type = frame_header[1];
+        payload_len = 0;
+        data_available(&frame.ptr[0], payload_len); // Notify DISC
+        frame.advance(payload_offset);
+        state = cmux_state::FOOTER;
+        return true;
+    }
if ((frame_header[3] & 1) == 0) {
      if (frame_header_offset + frame.len <= 4) {
            frame_header_offset += frame.len;
```

4) Device CAVLI C16QS does not correctly enter CMUX mode with esp_modem. The CMUX as defined in 3GPP TS 27.010: SABM response (paragraph 5.4.1) should be a UA frame (upon success, DM frame on failure). This device however responds with 0x3F, which is neither UA nor DM. You can apply the below patch to adapt the entry sequence

```
diff --git a/components/esp_modem/src/esp_modem_cmux.cpp
b/components/esp_modem/src/esp_modem_cmux.cpp
index c47e13b..7afbf73 100644
--- a/components/esp_modem/src/esp_modem_cmux.cpp
+++ b/components/esp_modem/src/esp_modem_cmux.cpp
@@ -137,7 +137,8 @@ bool CMux::data_available(uint8_t *data, size_t len)
      } else {
            return false;
      }
-    } else if (data == nullptr && type == (FT_UA | PF) && len == 0) { // notify the initial
SABM command
+    } else if (data == nullptr && (type == (FT_UA | PF) || type == 0x3f) && len == 0) { //
notify the initial SABM command
      Scoped<Lock> l(lock);
      sabm_ack = dlci;
} else if (data == nullptr && dlci > 0) {
@@ -238,8 +239,7 @@ bool CMux::on_header(CMuxFrame &frame)
type = frame_header[2];
// Sanity check for expected values of DLCI and type,
// since CRC could be evaluated after the frame payload gets received
-    if (dlci > MAX_TERMINALS_NUM || (frame_header[1] & 0x01) == 0 ||
-            (((type & FT_UIH) != FT_UIH) &&  type != (FT_UA | PF) ) ) {
+    if (dlci > MAX_TERMINALS_NUM) {
      recover_protocol(protocol_mismatch_reason::UNEXPECTED_HEADER);
      return true;
}
```