ZX-ESP WiFi internet card API

1. Ограничения и возможности

1.1 Ограничения ZX-SPECTRUM

С точки зрения нормальной работы по обмену данными, основным ограничением ZX-SPECTRUM является отсутствие прерываний от COM-порта.

В некоторых клонах такое прерывание вроде бы и есть, но в общем случае — его, увы нет. Это приводит к тому, что при разработке асинхронных систем обмена данными, а интернет-соединение - именно такая система в общем случае.

По-простому — асинхронная, значит, что данные после установления соединения могут прилететь в любой момент. И объём этих данных неизвестен.

Выход — постоянный опрос COM-порта. Чем это чревато — объяснять не надо. COM-порт устройство относительно медленное и ваша программа будет постоянно висеть и ждать - «а не пришел ли пакет с той стороны?», вместо того, чтобы заниматься чем-то полезным.

Разумеется, опрос можно повесить на прерывание и впадать в состояние ожидания только когда «на той стороне» будут данные. Это улучшит картину, но не сильно.

Пока смиримся с тем что есть.

1.2 Возможности ESP8266

Возможности ESP8266 весьма широки:

- поддержка режима WiFi-точки доступа или станции (можно и то и другое одновременно, но, говорят, не очень хорошо работает — сам не проверял);
- поддержка до 4х IP-соединений одновременно (для ZX хватит за глаза);
- поддержка слушающих и соединяющихся сокетов (то есть можно как коннектится к серверам, так и самому быть сервером);
- очень неплохой SDK, который позволяет на языках C/C++ реализовать многие сетевые фантазии и запихать их в вашу уникальную прошивку;
- встроенная периферия — в частности UART, он же - COM-порт.

Для нас важно, что мы можем реализовать удобный нам протокол с помощью этого SDK не оглядываясь на ограничения AT-команд стандартной прошивки.

2. Минимально необходимое API

Что нам необходимо для работы с сетью? А ничего выдумывать не надо. Можно просто поглядеть как это сделано в существующих системах и сделать упрощённую копию API со стороны пользователя. Уникальным будет только API для настройки WiFi.

Сразу оговорюсь, что скорость обмена по COM-порту будет фиксированной — 115200. Может потом её можно будет и изменять, но пока забью на эти заморочки.

Итак, минимальное API для настройки WiFi.

2.1 WiFi API

Функция настройки WiFi:

int8_t wifi_config(uint8_t mode, const char* name, const char* pass, uint8_t auth);

mode - режим (AP/Station)
name - имя точки доступа;
pass - пароль;
auth - режим аутентификации (только для AP).

Возвращает функция 0 если всё хорошо или код ошибки.

Функция проверки состояния WiFi:
int8_t wifi_status(uint8_t* mode, char* name, char* pass, uint8_t* auth);

Возвращает 0 если всё хорошо или код ошибки, так же возвращает:
mode - режим (AP/Station)
name - имя точки доступа;
pass - пароль;
auth - режим аутентификации (только для AP).

Для AP и Station режимов коды ошибок разные.

2.2 Socket API

Функции работы с сокетами. Я честно содрал их из юникса. Чтобы было:)

2.2.1. Общие функции

int8_t socket(int8_t domain, int8_t type, int8_t protocol);

Создаёт сокет.
domain — для IP — всегда AF_INET;
type — SOCK_STREAM для TCP и SOCK_DGRAM для UDP (для начала хватит);
protocol — всегда 0.
Возвращает номер соединения (>=0) или код ошибки (<0).

int8_t close(int8_t fd);

Закрывает сокет.
fd — номер сокета (>=0).
Возвращает 0, если все в порядке или код ошибки (<0).

void esp_poll();

Тот самый мерзкий вызов, который опрашивает состояние ESP8266 по COM-порту и вычитывает данные. Должен вызываться периодически. Может быть — в прерывании.

2.2.2. Функции соединяющегося сокета

int8_t connect(int8_t sockfd, const struct sockaddr *addr, int16_t addrlen);

Установить соединение.
sockfd — номер сокета, созданного функцией **socket()**;
sockaddr — адрес (для IP — это IP-адрес и номер порта);
addrlen — размер структуры sockaddr (он разный для разных сокетов, но у нас пока всегда 6 байт).
Возвращает 0, если соединение установлено или код ошибки, если нет.

int16_t recv(int8_t sockfd, void *buf, int16_t len, int8_t flags);

Читает данные из сокета.
sockfd — номер сокета;
buf — буфер, куда читать данные;
len — размер буфера;
flags — флаги (пока не используются).

Возвращает: количество считанных байт (>0), код ошибки (<0) или, если возвращен 0, то соединение закрыто.

int16_t send(int8_t sockfd, const void *buf, int16_t len, int flags);
Отправляет данные в сокет.
sockfd — номер сокета;
buf — буфер, с данными данные;
len — сколько байт нужно отправить;
flags — флаги (пока не используются).

Возвращает: количество переданных байт (>0), код ошибки (<0) или, если возвращен 0, то соединение закрыто.

2.2.3. Функции слушающего сокета (может быть потом и их реализуем)

int8_t bind(int8_t sockfd, const struct sockaddr *addr, int16_t addrlen);

int8_t listen(int8_t sockfd, int8_t backlog);

int8_t accept(int8_t sockfd, struct sockaddr *addr, int16_t *addrlen);

3. Протокол обмена

По сути, необходимо преобразовать асинхронный обмен данными в синхронный.

Управляющим устройством (master) будет ZX, а ESP8266 должно выполнять его команды.

Все принятые данные ESP8266 должно хранить в своих буферах, насколько позволит размер памяти.

Так как к COM-порту будет привязано не более одного устройства, то команды — безадресные.

В качестве транспорта используем протокол SLIP.

Границей SLIP-кадра является уникальный флаг END (0xC0). Уникальность этого флага поддерживается байт-стаффингом внутри кадра с ESC-последовательностью 0xDB, причём байт END (0xC0) заменяется последовательностью (0xDB, 0xDC), а байт ESC (0xDB) — последовательностью (0xDB, 0xDD).

Такой формат позволит точно найти начало и конец пакета (кадра) при сбое, накладные расходы на анализ байт — минимальны.

Общий формат команд (исходный, до байт-стаффинга):

Резерв, всегда 0	**Код команды**	**Длина данных (0-данных нет и CRC8 данных нет) NNNN**	**CRC8 заголовка**	**Данные (зависят от кода команды)**	**CRC8 данных**
1 байт	**1 байт**	**2байта**	**1 байт**	**NNNN байт**	**1 байт**
Заголовок					

Каждая команда предполагает ответ от ESP8266.