MeGaWiFi

WiFi connectivity for the SEGA Genesis/Megadrive console

Designed and implemented for 1985alternativo

by doragasu (2015 – 2016)

Index

Figure Index

Table Index

1 Introduction

MeGaWiFi is a programmable cartridge for the 16-bit console SEGA Genesis/Megadrive. It contains the hardware required to allow wireless WiFi connectivity. The cartridge is accompanied by a programmer. The programmer is connected to a PC to read and program the cartridge Flash memory chip containing the game ROM. The programmer also allows to upload custom firmwares for the WiFi module, allowing to test the UART and WiFi connectivity.
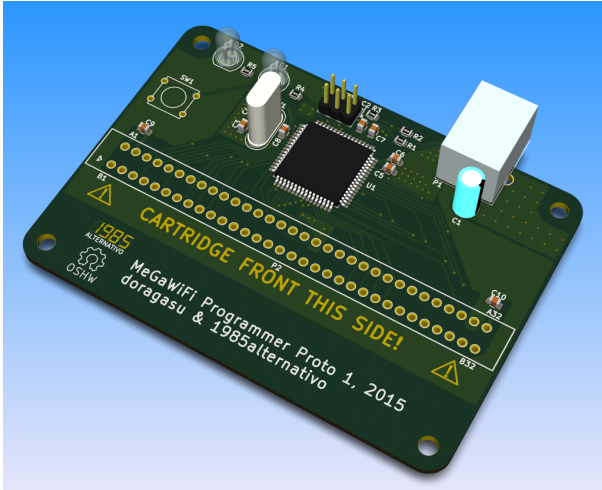
Figure 1: MeGaWiFi programmer
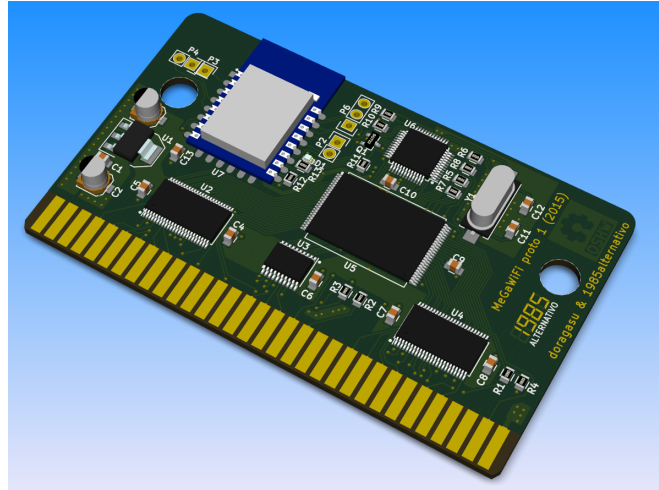
Figure 2: MeGaWiFi cartridge

The cartridge consists of the blocks shown on Figure 3.
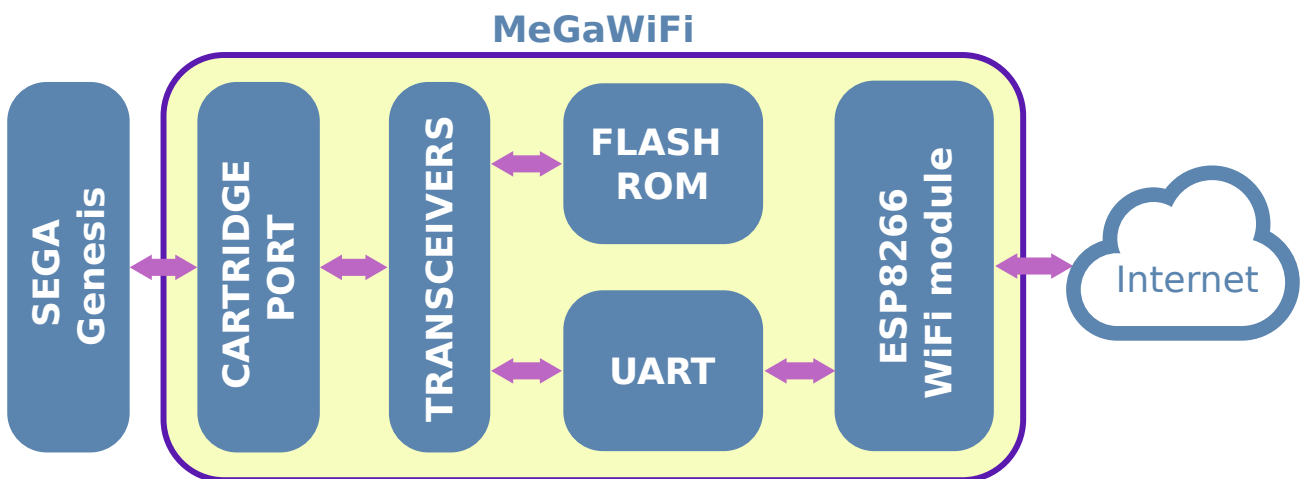
Figure 3: MeGaWiFi blocks

The cartridge is connected to the Genesis/Megadrive console using the Cartridge port. As the Megadrive console uses 5V TTL levels, and the Flash chip, the UART and the WiFi module are 3.3V CMOS devices, the cartridge includes some transceivers to adapt voltage levels.

The Flash Rom block is directly connected to the console memory buses. This way the console can read and execute the game stored inside the Flash.

To handle wireless communications, an ESP8266 WiFi module is used. This module consists of

a 32-bit microcontroller and the required wireless interface to implement WiFi communications, including the physical, link, network and transport layers. The microcontroller inside the ESP8266 module can also run user programs, allowing it to help the console implementing the network logic.

As the ESP8266 module cannot be directly connected to the console buses, it is necessary to include an UART block, that will act as a bridge between the console memory buses and the ESP8266 module.

The programmer only has 3 components: an USB interface to connect with the host PC, an AT90USB646 microcontroller, and the cartridge interface used to dialog with the cartridge. The microcontroller runs a firmware that controls the USB and cartridge interfaces, and implements a protocol allowing the PC to read and program the flash chip.

2 Cartridge programmer software

The cartridge programmer software consists of two components: the program running on the PC (from now on, the PC application, or just the application), and the firmware running on the microcontroller of the programmer (from now on, the programmer firmware, or just the firmware). Both software components can communicate using a simple protocol.

2.1 MegaDrive Memory Administration Protocol (MDMAP)

To be able to send and receive ROMs to/from the programmer, a very simple protocol has been defined. The protocol is implemented on top of the USB stack, using vendor-defined USB bulk transfers.

> ⚠️ **WARNING**: unless otherwise specified, byte order for parameters larger than 1 octet, is LITTLE ENDIAN.

MDMAP is a typical command/response protocol. The programmer firmware acts as a server, and the applications as a client. The application sends commands to the firmware. The firmware receives these commands, executes them, and sends back a response to the application. Each time the client sends a command, it must wait for the response before issuing another command.

Table 1 shows currently supported commands. To simplify implementation, commands and responses are defined both on the same table.

Table 1: Supported MDMAP commands

Code	Command	Description
0	MDMA_OK	Used to report OK status during command replies
1	MDMA_MANID_GET	Flash chip manufacturer ID request
2	MDMA_DEVID_GET	Flash chip device ID request
3	MDMA_READ	Flash read request
4	MDMA_CART_ERASE	Entire flash chip erase request
5	MDMA_SECT_ERASE	Flash chip sector erase request
6	MDMA_WRITE	Flash chip program request
7	MDMA_MAN_CTRL	Manual GPIO control request (dangerous!, TBD)
8	MDMA_BOOTLOADER	Enters DFU bootloader mode, to update MDMA firmware
9	MDMA_BUTTON_GET	Obtains the programmer pushbutton status.
10	MDMA_WIFI_CMD	Forward command to the WiFi module.
11	MDMA_WIFI_CMD_LONG	Forward comand with long data payload to WiFi module.
12	MDMA_WIFI_CTRL	Execute a control action on the WiFi module.
255	MDMA_ERR	Used to report ERROR status during command replies

All commands and responses have at least one byte (the command or CMD field). Depending on the command implementation, it may have more fields. Although most of the commands are sent using a single USB bulk transfer, some commands and responses might span for two transfers. As an example, the MDMA_WRITE command, whose format is shown in Figure 4, is sent using two transfers, the first one contains the command code, payload length and payload address. The second one contains the data payload. The format of the command responses is exactly the same, but instead of a command, the first byte will contain a response code (MDMA_OK or MDMA_ERR). As happens with command requests, command responses can have additional fields, and although most of them are completed on a single USB bulk transfer, some can take up to two transfers to complete.
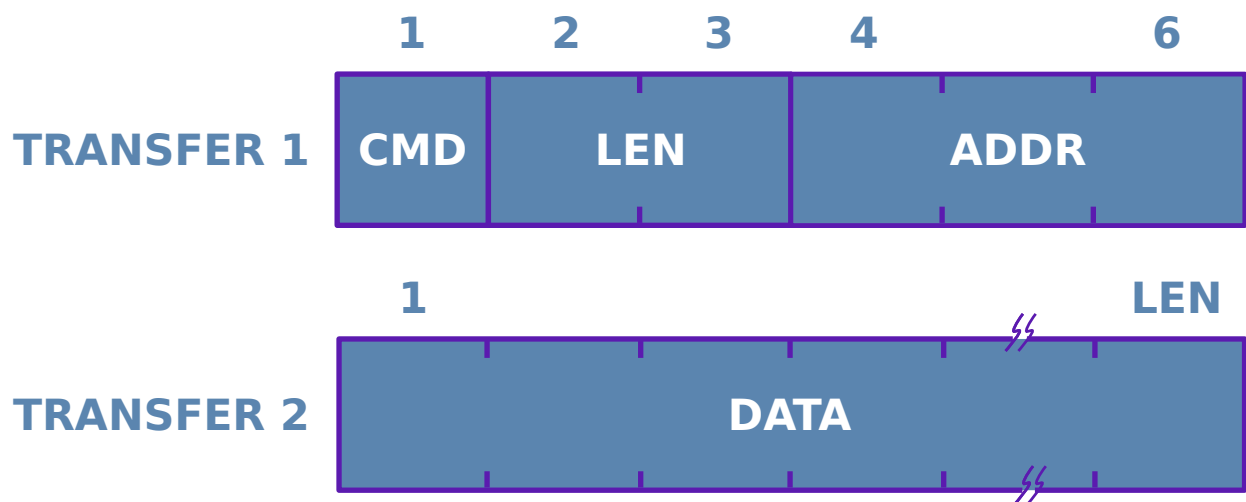
Figure 4: MDMA_WRITE command

The following sections describe USB and command implementation details.

2.1.1 USB requirements

The USB interface implements a Full Speed Bulk pipe, with an IN endpoint and and OUT one (plus the mandatory control endpoint). Details for this interface are shown on Table 2.

Table 2: USB interface details

USB Specification	1.1
Speed	Full (12 Mbps)
Vendor ID	0x03EB
Product ID	0x206C
Class	Vendor defined
IN endpoint address	0x83
OUT endpoint address	0x04
Max power consumption	500 mA[1]
Polling interval (ms)	1 ms
Bulk IN endpoint length	64 bytes
Bulk OUT endpoint length	64 bytes

2.1.2 Command specification

MDMA_MANID_GET

Obtains the flash memory chip manufacturer ID.

- Size: 1 byte.

- Format: MDMA_MANID_GET (1 byte).

- Reply:

  - MDMA_OK: Manufacturer ID obtained successfully. *manID* contains the manufacturer ID code.

    - Size: 3 bytes.

    - Format: MDMA_OK (1 byte) + *manID* (1 word).

  - MDMA_ERR: Error while obtaining manufacturer ID.

    - Size: 1 byte.

    - Format: MDMA_ERR (1 byte).

MDMA_DEVID_GET

Obtains the flash memory chip device ID.

- Size: 1 byte.

- Format: MDMA_DEVID_GET (1 byte).

- Reply:

---

1   Not based on measurements, real power consumption should be much lower.

- ○ MDMA_OK: Device ID obtained successfully. *devID* array contains device ID codes.
    - ▪ Size: 7 bytes.
    - ▪ Format: MDMA_OK (1 byte) + *devID* (3 codes, 1 word each):
- ○ MDMA_ERR: Error while obtaining device ID.
    - ▪ Size: 1 byte.
    - ▪ Format: MDMA_ERR (1 byte).

MDMA_READ

Reads data from the flash memory chip. *wLen* parameter contains the number or words to read, and *addr* parameter contains the address at which the read operation is performed. Data is sent using up to two transfers. The first transfers contains the command response, including command status. If status code is MDMA_OK, a second transfer contains the requested data payload.

- • Size: 6 bytes.
- • Format: MDMA_READ (1 byte) + *wLen* (2 bytes) + *addr* (3 bytes).
- • Reply:
    - ○ MDMA_OK: Data read operation completed successfully. *wLen* parameter contains the number of words read, *addr* the initial address for the read operation, and *data* the readed data.
        - ▪ Size: 7 to 38 bytes.
        - ▪ Format: MDMA_OK (1 byte) + *wLen* (1 byte) + *addr* (1 dword) + *data* (1 to 16 words in BIG ENDIAN byte order).
    - ○ MDMA_ERR: Could not perform read operation.
        - ▪ Size: 1 byte.
        - ▪ Format: MDMA_ERR (1 byte).

Note: *wLen* must be between 1 and 16, or the read operation will fail.

Warning: returned data is in BIG ENDIAN byte order.

MDMA_CART_ERASE

Erases the entire flash chip contents.

- • Size: 1 byte.
- • Format: MDMA_CART_ERASE (1 byte).
- • Reply:

- ○ MDMA_OK: Flash chip entirely erased.
  - ▪ Size: 1 byte.
  - ▪ Format: MDMA_OK (1 byte).
- ○ MDMA_ERR: Could not erase entire flash chip.
  - ▪ Size: 1 byte.
  - ▪ Format: MDMA_ERR (1 byte).

Note: On flash chips, cleared data bytes are read as 0xFF.

MDMA_SECT_ERASE

Erases one sector of the flash chip. Erased sector is the one in which provided *addr* address parameter falls in.

- • Size: 5 bytes.
- • Format: MDMA_SECT_ERASE (1 byte) + *addr* (1 dword).
- • Reply:
  - ○ MDMA_OK: Flash chip sector erased.
    - ▪ Size: 1 byte.
    - ▪ Format: MDMA_OK (1 byte).
  - ○ MDMA_ERR: Could not erase flash chip sector.
    - ▪ Size: 1 byte.
    - ▪ Format: MDMA_ERR (1 byte).

MDMA_WRITE

Programs (writes) data to the specified address range. The programmed address range should be cleared, or written data will most likely be corrupt. *addr* parameter is the address to which data will be written, *wLen* is the number of words to write, and *data* is the array of words to write to the flash memory chip. Command is sent split on two transfers. The first one sends the command request, and the second transfer sends the data payload.

- • 1st transfer size: 6 bytes.
- • 1st transfer format: MDMA_WRITE (1 byte) + *wLen* (2 bytes) + *addr* (3 bytes).
- • 2nd transfer size: 2 to 65534 bytes (1 to 32767 words).
- • 2nd transfer format: Write data payload, in BIG ENDIAN byte ordering.
- • Reply:

- ○ MDMA_OK: *data* correctly written to *addr*.
  - ■ Size: 1 byte.
  - ■ Format: MDMA_OK (1 byte).
- ○ MDMA_ERR: Could not write requested data.
  - ■ Size: 1 byte.
  - ■ Format: MDMA_ERR (1 byte).

Note: Due to performance reasons, it should be avoided to cross write-buffer boundaries during writes. This can be guaranteed by checking that during a MDMA_WRITE command, the 4 lower address bits do not cross beyond 1111b (going back to 0000b). E.g. if we do a 16 word or longer transfer, the only way to guarantee we are not crossing a write-buffer boundary, is by checking that the 4 lower bits of *addr* field, are 0. If we do a 8 byte transfer, the 4 lower bits of *addr* can be from 0 to 7.

Warning: data payload must be sent in BIG ENDIAN byte order.

MDMA_MAN_CTRL

Allows to manually control the microcontroller GPIO pins.

- • Size: 24 bytes.
- • Format: as shown on table 3.
- • Reply:
  - ○ MDMA_OK: Data successfully readed and/or written to specified ports pins.
    - ■ Size: 7 bytes.
    - ■ Format: MDMA_OK (1 byte) + readed port data (6 bytes).
  - ○ MDMA_ERR: Data could not be readed/written.
    - ■ Size: 1 byte.
    - ■ Format: MDMA_ERR (1 byte).

Table 3: Manual GPIO control command format.

UNLOCK SEQ:	CMD	19	85	BA	DA	55
PIN MASK:	PA	PB	PC	PD	PE	PF
R/$\overline{\text{W}}$:	PA	PB	PC	PD	PE	PF
VALUE:	PA	PB	PC	PD	PE	PF

Each cell on table 3 represents one byte of the command, so each row has six bytes. On the first

row the command (CMD, MDMA_MAN_CTRL) is set, along with the unlock sequence (5 bytes with the hexadecimal values 1985BADA55).

The next 3 lines on table 3 contain the data needed to complete the requested GPIO action for each bit of the 6 available port pins (PA, PB, PC, PD and PE). Line 2 has the pin mask: any bits set will cause the corresponding port pin to be readed or written.

Line 3 has the read/write atribute: for each of the enabled pins on the pin mask line, a R/$\overline{W}$ value of 1 means the pin will be readed, and a value of 0 means the pin will be written to.

Line 4 is only used when writing data to port pins (although this line must always be sent even if not writing to any pins). The contents on this line are written to the pins enabled on the pin mask.

Note: Reads are performed **before** writes.

> ⚠️ **WARNING**: this command is dangerous! If used without care, it might damage the cartridge and/or the programmer. Extreme precaution must be observed, specially when writing to port pins!

Warning: this function is dangerous! If used without care, it might damage the cartridge and/or the programmer. Extreme precaution must be observed, specially when writing to port pins!

MDMA_BOOTLOADER

Enters DFU bootloader mode, allowing to update MDMA firmware (using e.g. *dfu-programmer* software).

- Size: 1 byte.

- Format: MDMA_BOOTLOADER (1 byte).

Note: This command is not replied. When acknowledged, the device detaches the USB interface immediately and enters DFU mode about 2 seconds later. Client software should not expect this command to be replied.

MDMA_BUTTON_GET

Reads the programmer pushbutton status. The reply to this command contains the pushbutton status (pressed/released) and if a pushbutton event has occurred since the last time this command was issued.

- Size: 1 byte.

- Format: MDMA_BUTTON_GET (1 byte).

- Reply:

  - MDMA_OK: pushbutton status obtained.

    - Size: 2 bytes.

- Format: MDMA_OK (1 byte) + button status (1 byte).

Button status is reported as follows (only the 2 least significant bits are used, the remaining 6 bits should be ignored):

- BIT0: pushbutton status. The pushbutton is pressed if this bit is set.

- BIT1: pushbutton event. If this bit is set, there has been an event (button press and/or release) since the last time this command was issued. Note this bit is reset each time the programmer replies to this command.

MDMA_WIFI_CMD

Forward a short command (up to 60 bytes) to the WiFi module.

- Size: variable (4 ~ 64 bytes).

- Format: MDMA_WIFI_CMD (1 byte) + command length (2 bytes) + padding (1 byte) + command data (up to 60 bytes).

- Reply:

  ○ MDMA_OK: Command successfully forwarded.

    - Size: variable (depends on command sent to WiFi module).

    - Format: MDMA_OK + WiFi command reply bytes (2 ~ N, first byte is no returned).

  ○ MDMA_ERROR: Could not forward the command to the WiFi module.

    - Size: 1 byte.

    - Format: MDMA_ERROR (1 byte).

MDMA_WIFI_CMD_LONG

Forward a long command (more than 60 bytes) to the WiFi module. Command must be sent on two transfers. The first transfer sends the command and the data length. The second transfer sends the command data forwarded to the WiFi module.

- Size: N bytes.

- 1st transfer format: MDMA_WIFI_CMD (1 byte) + command length (2 bytes) + padding (1 byte).

- 2nd transfer format: command data (1 to 65535 bytes).

- Reply:

  ○ MDMA_OK: Command successfully forwarded.

    - Size: variable (depends on command sent to WiFi module).

- Format: MDMA_OK + WiFi command reply bytes (2 ~ N, first byte of the WiFi module command reply is replaced by MDMA_OK).
  - MDMA_ERROR: Could not forward the command to the WiFi module.
    - Size: 1 byte.
    - Format: MDMA_ERROR (1 byte).

MDMA_WIFI_CTRL

Sends a WiFi module control command. Supported control commands are shown on table 4.

- Size: 2 or 3 bytes.

- Format: MDMA_WIFI_CTRL (1 byte) + control command (1 byte) + control command data (1 byte, only uased for WIFI_CTRL_SYNC control command).

- Reply:

  - MDMA_OK: Control action successfully executed.

    - Size: 1 byte.

    - Format: MDMA_OK (1 byte).

  - MDMA_ERROR: Could not execute control action.

    - Size: 1 byte.

    - Format: MDMA_ERROR (1 byte).

Table 4: Supported WiFi module control commands

Command	Data	Description
WIFI_CTRL_RST	No	Puts WiFi module in RESET.
WIFI_CTRL_RUN	No	Releases RESET from WiFi module.
WIFI_CTRL_BLOAD	No	Sets GPIO pins to start in bootloader mode.
WIFI_CTRL_APP	No	Sets GPIO pins to start in application (normal) mode.
WIFI_CTRL_SYNC	1 byte	Performs a SYNC cycle. Requires a data byte with the number of sync retries.

2.2 Host PC application

The host PC application connects to the programmer to read and program the cartridge contents and to program the WiFi module firmware. The PC application provides a command line interface, to be able to integrate it with cartridge flash scripts and GUI applications as needed.

2.2.1 Command line interface

The command line application invocation must be as follows:

```
mdma [option1 [option1_arg]] […] [optionN [optionN_arg]]
```

The options (*option1 ~ optionN*) can be any combination of the ones listed in table 5. Options must support short and long formats. Depending on the used option, and option argument (*option_arg*) must be supplied. Several options can be used on the same command, as long as the combination makes sense (e.g. it does make sense using the flash and verify options together, but using the help option with the flash option doesn't make too much sense).

Table 5: Command line interface options

Option (long, short)	Argument type	Description
`--flash, -f`	R - File	Programs the contents of a file to the cartridge flash chip.
`--read, -r`	R - File	Read the flash chip, storing contents on a file.
`--erase, -e`	N/A	Erase entire flash chip.
`--sect-erase, -s`	R - Address	Erase flash sector corresponding to address argument.
`--verify, -V`	N/A	Verify written file after a flash operation.
`--flash-id, -i`	N/A	Print information about the flash chip installed on the cart.
`--gpio-ctrl, -g`	R - Pin data	Manually control GPIO port pins of the microcontroller.
`--wifi-flash, -w`	R - File	Uploads a firmware blob to the cartridge WiFi module.
`--pushbutton, -p`	N/A	Read programmer pushbutton status.
`--bootloader, -b`	N/A	Enters DFU bootloader mode, to update programmer firmware.
`--dry-run, -d`	N/A	Performs a dry run (parses command line but does nothing).
`--version, -R`	N/A	Print version information and exit.
`--verbose, -v`	N/A	Write additional information on console while performing actions.
`--help, -h`	N/A	Print a brief help screen and exit.

The *Argument type* column in table 5 contains information about the parameters associated with every option. If the option takes no arguments, it is indicated by "N/A" string. If the option takes a required argument, the argument type is prefixed with "R" character. Supported argument types are *File, Address* and *Pin Data*:

- *File*: Specifies a file name. Along with the file name, optional *address* and *length* fields can be added, separated by the colon (:) character, resulting in the following format:

```
file_name[:address[:length]]
```

- *Address*: Specifies an address related to the command (e.g. the address to which to flash a cartridge ROM or WiFi firmware blob).

- *Pin Data*: Data related to the read/write operation of the port pins, with the format:

```
pin_mask:read_write[:value]
```

When using *Pin Data* arguments, each of the 3 possible parameters takes 6 bytes: one for each 8-bit port on the chip from PA to PF. Each of the arguments corresponds to the row with the same name on table 3. The *value* parameter is only required when writing to any pin on the ports. It is recommended to specify each parameter using hexadecimal values (using the prefix '0x').

The `--pushbutton` switch returns pushbutton status on the program exit code (so it is easily readable for programs/scripts using *mdma-cli*. The returned code uses the two least significant bits:

- BIT0: pushbutton status. The pushbutton is pressed if this bit is set.

- BIT1: pushbutton event. If this bit is set, there has been an event (button press and/or release) since the last *mdma-cli* invocaton. Note this bit is reset each time the program is launched with the `--pushbutton` switch.

E.g. if the button is pressed, and keeps being pressed when the program evaluates the `--pushbutton` function, the returned code will be 0x03 (pushbutton event + button pressed). If immediately called before the button is released, returned code will be 0x01 (no event + button pressed). If the button is released and then the program is called again, returned code will be 0x02 (pushbutton event + no button pressed).

Some more examples of the command invocation and its arguments are:

- `mdma -ef rom_file` → Erases entire cartridge and flashes *rom_file*.

- `mdma --erase -f rom_file:0x100000` → Erases entire cartridge and flashes contents of *rom_file*, starting at address 0x100000.

- `mdma -s 0x100000` → Erases flash sector containing 0x100000 address.

- `mdma -Vf rom_file:0x100000:32768` → Flashes 32 KiB of *rom_file* to address 0x100000, and verifies the operation.

- `mdma --read rom_file::1048576` → Reads 1 MiB of the cartridge flash, and writes it to *rom_file*. Note that if you want to specify *length* but do not want to specify *address*, you have to use two colon characters before *length*. This way, missing *address* argument is interpreted as 0.

- `mdma -g 0xFF00FFFF0000:0x110000000000:0x000012340000` → Reads data on port A, and writes 0x1234 on ports PC and PD.

- `mdma -w wifi-firm:0x40000` → Uploads *wifi-firm* firmware blob to the WiFi module, at address 0x40000.

3 MeGaWiFi firmware for ESP8266

MeGaWiFi cartridges contain an ESP8266 wireless module running a custom firmware, to handle wireless communications, Internet protocols and help implementing game network logic. This firmware communicates directly with the 68000 inside the console through the in-cart UART.

Although the firmware can be customized for every application, the default implementation consists of a server waiting for the client to issue commands. The communication flow is command/response oriented, but can change once communications are established, when using *transparent mode*. *Transparent mode* can be enabled once the module has established a TCP connection, or has configured an UDP socket. When in *transparent mode*, the command interpreter inside the WiFi chip is disabled, and the WiFi module acts as a transparent bridge, routing received data from the 68000 to the configured network end, and routing data received through the Internet to the 68000 host. *Transparent mode* ends when the TCP connection is closed by the other end, or using a special handshake mechanism provided by physical GPIO lines.

The firmware is based on esp-open-rtos, an open fork of the original toolchain developed by Espressif. This firmware uses lwIP for the network stack, and FreeRTOS for the multitasking support. These two components are two well tested solutions, often used in the embedded world.

Although a default implementation will be documented and developed, the open nature of the firmware allows it to be modified, to suit the game or application developed for the console. The default implementation should be generic enough to cover the needs of most games/applications. The documentation will cover only the development of the generic default firmware.

3.1 Use cases

The following subsections briefly introduce the firmware functions by showing some use cases.

3.1.1 Access point management
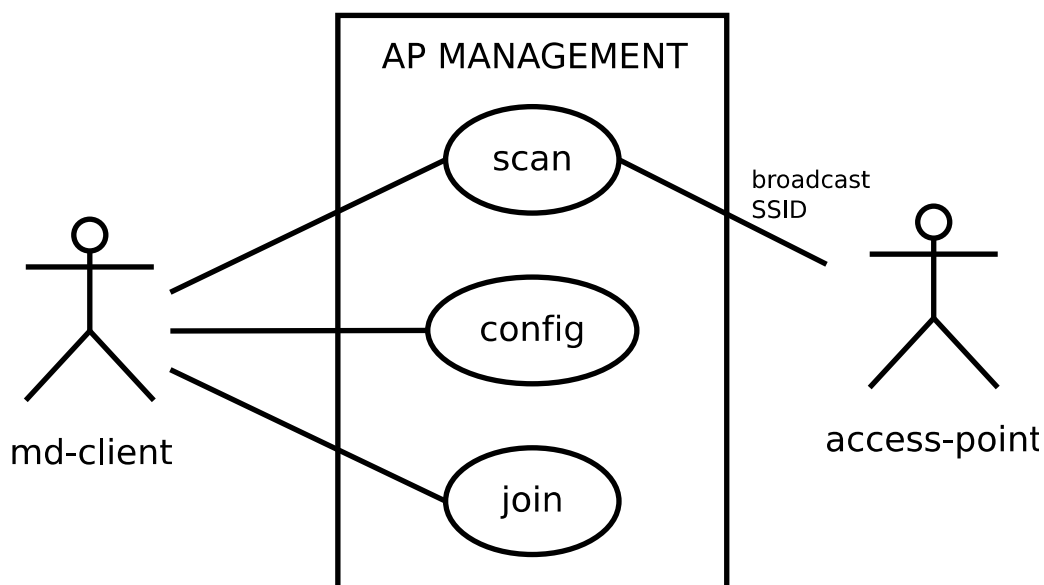
Figure 5: Access point management

- Actors:

1. **md-client** (initiating): software client that runs on the Megadrive CPU.

2. **access-point**: One or more WiFi access points.

- Pre-requisites: None.

The *access-points* (APs) are continuously broadcasting their SSIDs, along with some more additional information. The firmware must collect this information and provide means to *scan* the available SSIDs, *configure* connection information (SSID, authentication, IPv4 address, subnet mask, gateway, DNS). Minimal configuration consists of the SSID and authentication (if AP is not open).

Once configured, the md-client can request to *join* an AP.

3.1.2 TCP sockets

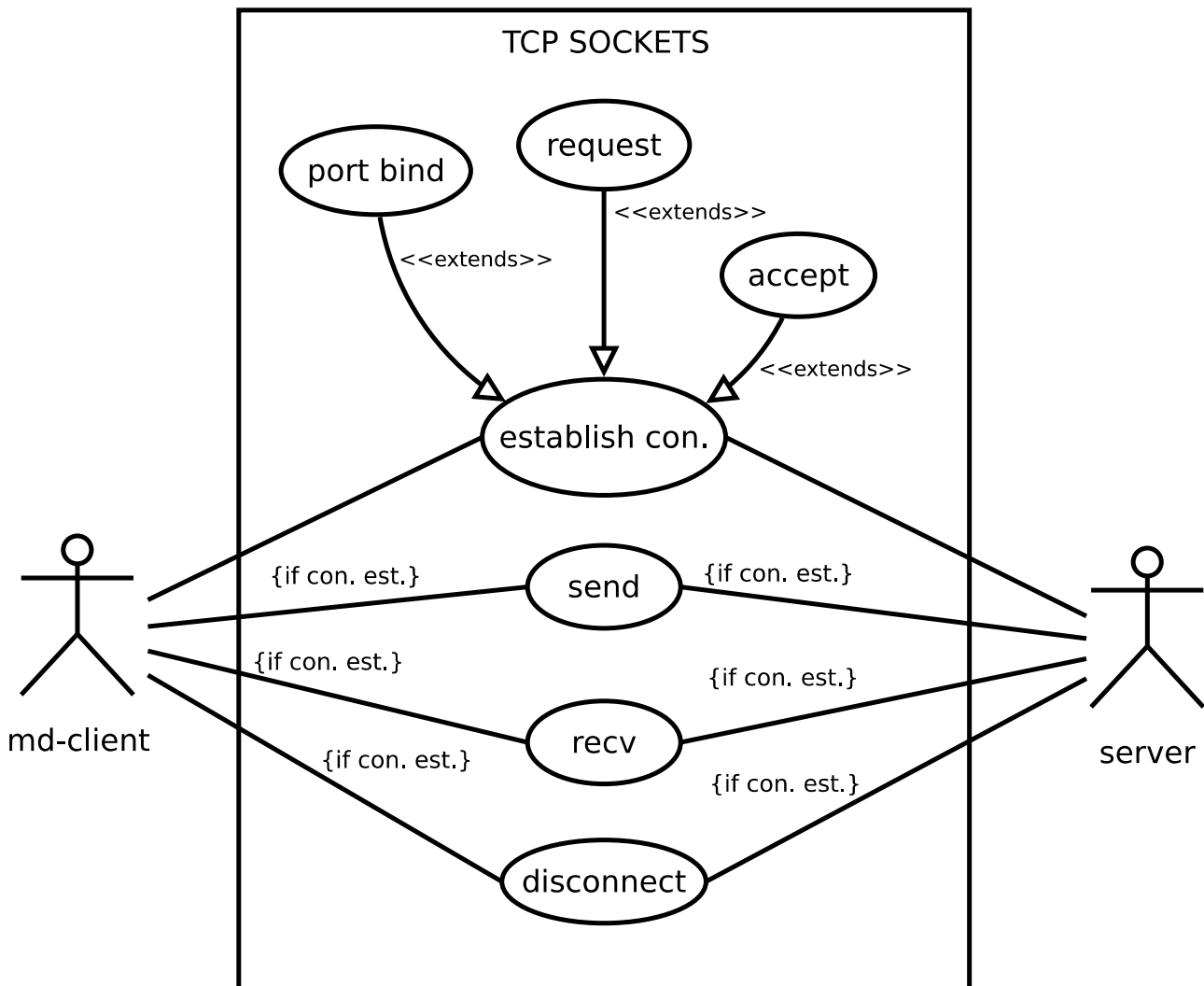*Figure 6: TCP sockets*

- Actors:

1. **md-client**: software client that runs on the Megadrive CPU.

2. **server**: machine providing services.

- Pre-requisites: the module must have joined an AP, as introduced on subsection 3.1.1.

- Note: Usually the md-client is the initiating actor, and connects to server, but on some scenarios, this roles can be reversed. Note that reversing the roles can be problematic for gaming applications, since UPnP is not supported, thus requiring the users to manually open/forward ports on their routers.

The firmware must provide means to communicate through The Internet using TCP protocol. *Establish connection* allows to initiate client connections using the *request* function. It also allows to create server sockets using *port bind*. Incoming connections on bound ports can be accepted using *accept* function.

Once a connection is established, the firmware must act as a bridge, so when the md-client *sends* data, it is forwarded to the server. Also the md-client can *receive* data sent by the server.

3.1.3 UDP sockets
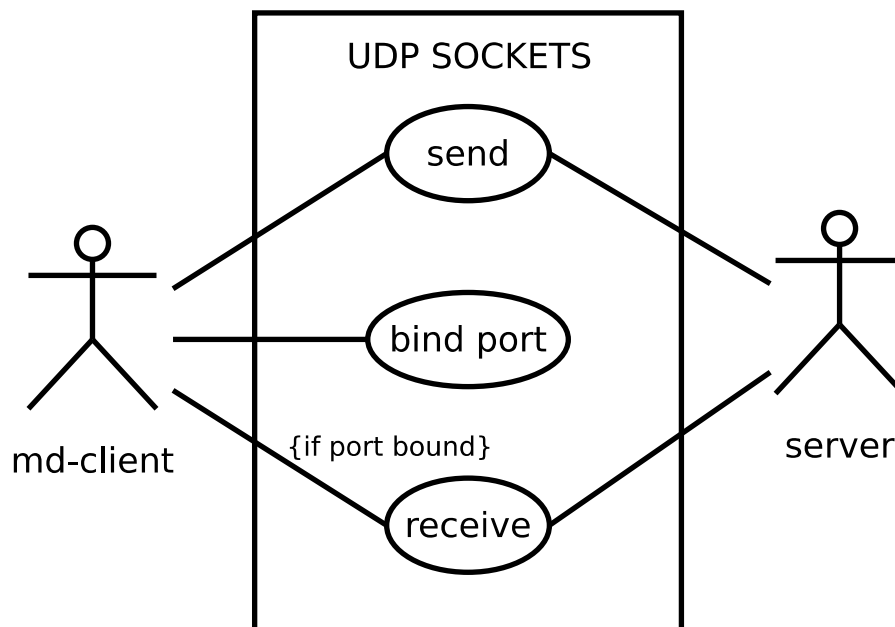
Figure 7: UDP sockets

- Actors:

1. **md-client**: software client that runs on the Megadrive CPU.

2. **server**: machine providing services.

- Pre-requisites: the module must have joined an AP, as introduced on subsection 3.1.1.

- Note: Receiving data using UDP can be problematic, because UPnP is not supported and users must manually open/forward ports on their routers.

The firmware must provide means to send and receive data using UDP protocol. As UDP is non connection-oriented, md-client can *send* data to the server at any moment. To receive data, the *bind*

port function must be used first. Once a port is bound, data can be *received* through it.

3.1.4 Misc functions
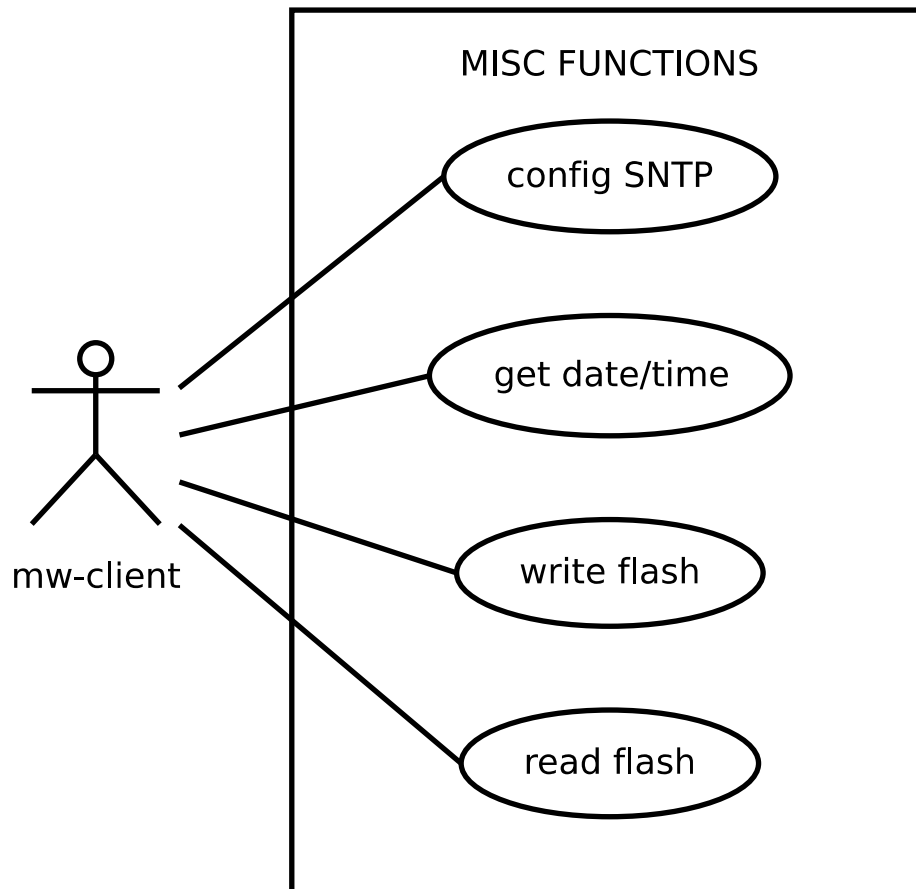
Figure 8: Misc functions

- Actors:

1. **md-client**: software client that runs on the Megadrive CPU.

The firmware must support other non-network related misc. functions:

- *config SNTP*: Configures SNTP parameters: NTP server URLs (up to 4) and update interval (greater than 15 s).

- *get date/time*: Obtains date and time. Values will most likely be wrong until SNTP has successfully updated the time.

- *write flash*: Allows to write to the flash chip inside the WiFi module.

- *read flash*: Allows to read from the flash chip inside the WiFi module.

3.2 Communications flow

Using MeGaWifi, involves the communications of at least 4 elements, as shown on Figure 9.

Figure 9: MeGaWiFi Communications flow

- **Megadrive console**: The console running usually a game.

- **WiFi module**: The WiFi module inside the MeGaWiFi cartridge, that provides the Megadrive console with WiFi connectivity.

- **Internet**: The Internet, including the access point used by the WiFi module, and whatever physical layer is used on the server side.

- **Server**: The server providing the service needed by the application running in the Megadrive console (e.g. online gaming software, online rankings, downloadable content, etc.).

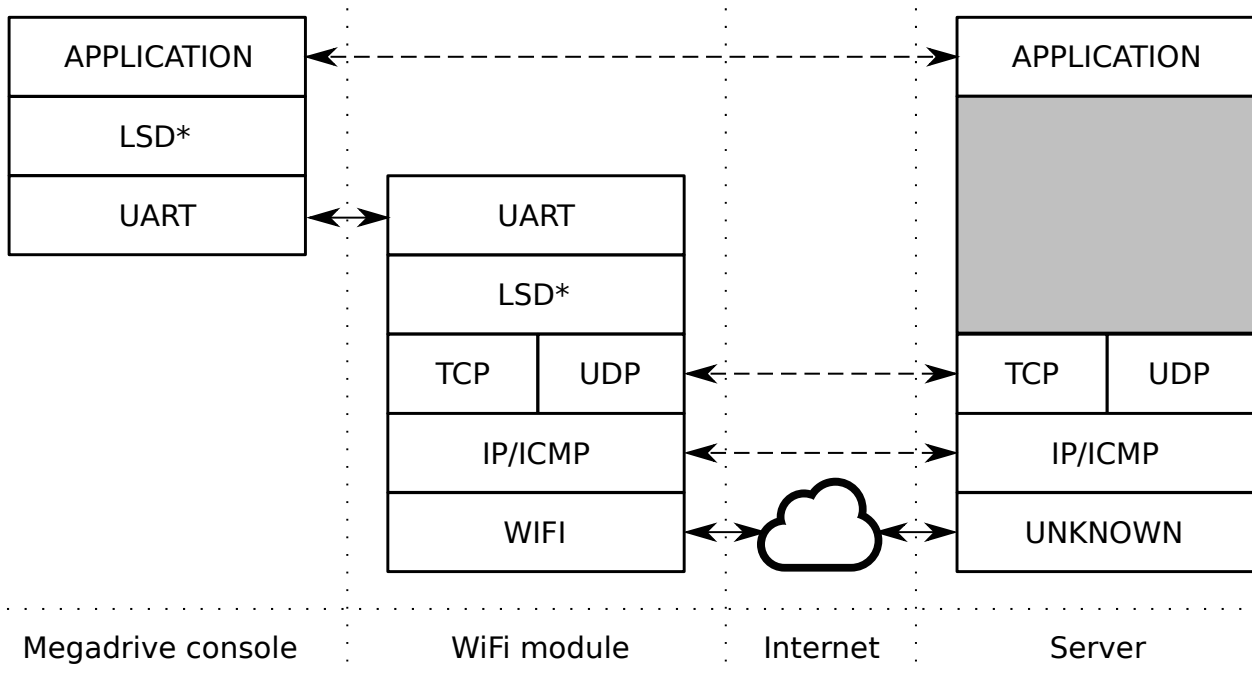Usually the application in the *Megadrive console*, needs to *talk* with the application running in the *Server*. As the *Server* is connected to The *Internet*, the communication must be facilitated by another element: the *WiFi module*. The WiFi module is accessed by the *Megadrive console* using a specially designed interface and protocol that will be further explained.

The protocol layers used to communicate through The *Internet* are standard. They are usually referred as the TCP/UDP/IP stack. Any device that wants to send/receive data through The Internet, must implement these protocols (or at the very least the IP protocol and another transport or application protocol). On the *Server* side, there is standard hardware (e.g. an Ethernet interface on a PC) and software (an OS such as GNU/Linux), providing means of using these protocols (usually as a *sockets* API), so development on it will not be covered in this document.

On the other side, the combination of the Megadrive console and the MeGaWiFi cartridge must implement these protocols, and allow the application running on the Megadrive CPU to use them, exposing a *BSD sockets* like interface.

As shown on Figure 9, the firmware running in the *WiFi module* implements the aforementioned

protocols. This is easily accomplished by using the esp-open-rtos framework for the ESP8266 WiFi module. As this framework includes lwIP network stack (implementing all the required protocols), we only need a way to expose these protocols, making them visible to the software running on the *Megadrive console*. Unfortunately this cannot be done directly, because the communication between the *Megadrive console* and the *WiFi module* is implemented using an UART, with limited bandwidth. Unless using an additional crystal oscillator on the MeGaWiFi cartridge (not used at least on the initial prototypes), maximum baud rate is limited for NTSC machines to:

$$BR_{NTSC,max} = \frac{7670000}{16} = 479375\,bps$$

On PAL machines, maximum baud rate is a bit lower:

$$BR_{PAL,max} = \frac{7610000}{16} = 475625\,bps$$

As the bandwidth is low, the method used to communicate the *Megadrive console* with the *WiFi module* must add the least possible overhead, to avoid further reducing the bandwidth.

> ⚠️ **WARNING**: At the time of writing this document, the UART link has only been successfully tested on a PAL machine with a 115200 baud rate. Further tests must be done to make sure the above mentioned speeds are achievable.

With these restrictions in mind, a simple protocol (LSD, Local Symmetric Data-link) has been designed to link the *Megadrive* console with the *WiFi module*, exposing the TCP and UDP protocols to the applications running on the *Megadrive* console CPU. The LSD protocol layer present on both the *Megadrive* program and the *WiFi module* firmware, along with the UART physical layer also present on both elements, acts as a bridge, allowing the software running on the *Megadrive* console to access the network stack implemented on the *WiFi module*. This way, a program running on the *Megadrive*, can send and receive data through *The Internet*. The LSD layer also implements a state machine allowing to handle commands and data redirection from/to the *Megadrive* (this is the reason for the LSD layer to be named with a trailing asterisk on Figure 9).

3.2.1 LSD Protocol

LSD (Local Symmetric Data-link) is a link layer (level 2) protocol designed with the following requirements:

- Operation over full-duplex serial links.

- Minimal protocol overhead.

- Physical channel multiplexing capabilities.

- Hardware flow control.

- TBD: Endianness awareness?

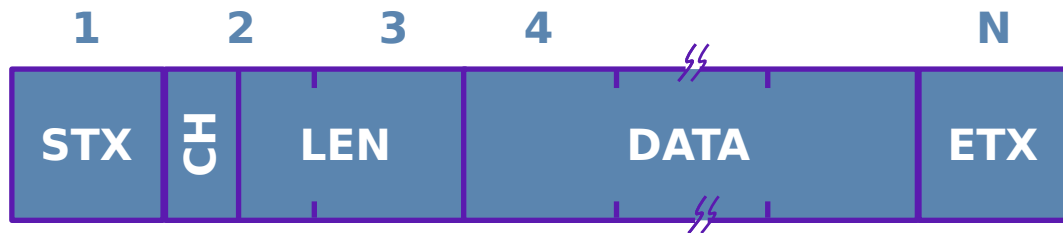LSD protocol data units (PDUs) have the format shown on Figure 10:

Figure 10: LSD PDU format

- Start of frame is signaled using `0x7F` STX character (1 byte).

- The byte following STX has two fields:

  - Channel number (CH) is stored on the high nibble of the second byte. It is used for physical link multiplexing, allowing up to 16 channels to operate on a single physical line.

  - The 4 upper bits of the payload length are stored on the lower nibble of the second byte.

- The third byte contains the lower 8 bits of the payload length. As payload length is 12 bit long, maximum payload length is 4096 bytes.

- Following the payload length, the payload data is sent. The number of bytes of the payload must match exactly the value specified for payload length.

- Finally, the `0x7F` ETX character must signal the end of frame.

To keep CPU requirements and overhead as low as possible, this protocol has no ECC mechanisms (checksum, CRC, retries, etc.). So it relies either on a reliable serial link, or on the ECC mechanisms implemented on upper layers.

To avoid overrun conditions, this protocol must implement hardware flow control, based on UART RTS/CTS lines.

This protocol is used to link the Megadrive console with the WiFi module. Channel multiplexing is used to have a separate channel for sending commands to the module firmware, and several additional channels reserved for socket usage. Unless working in *transparent mode*, data sent to channel 0 will be processed by the state machine implemented on the WiFi module, while data sent to any other channel will be transparently redirected to its corresponding socket (if the socket is active). Data received by the WiFi module on a previously opened socket, will be transparently redirected to its corresponding channel.

3.2.2 Principle of operation

Although the *WiFi module* can do a lot more tasks, its main purpose is providing a bridge between the *Megadrive* and a remote server. It accomplishes it by exposing to the *Megadrive* the network stack implemented in its firmware. Figure 11 shows a typical message sequence, establishing a connection with a remote *Server*, sending and receiving some data, and finally closing the connection. Messages between the *Megadrive* and the *WiFi module* are sent over the

serial line using LSD protocol. On the other hand, communication between the *WiFi module* and the *Server* are sent through the Internet, using sockets (with TCP/UDP and IP protocols). The diagram in Figure 11 does not show the internal TCP messages (such as SYN, ACK, etc.) to present a cleaner view, and because TCP implementation discussion is not the scope of this document.
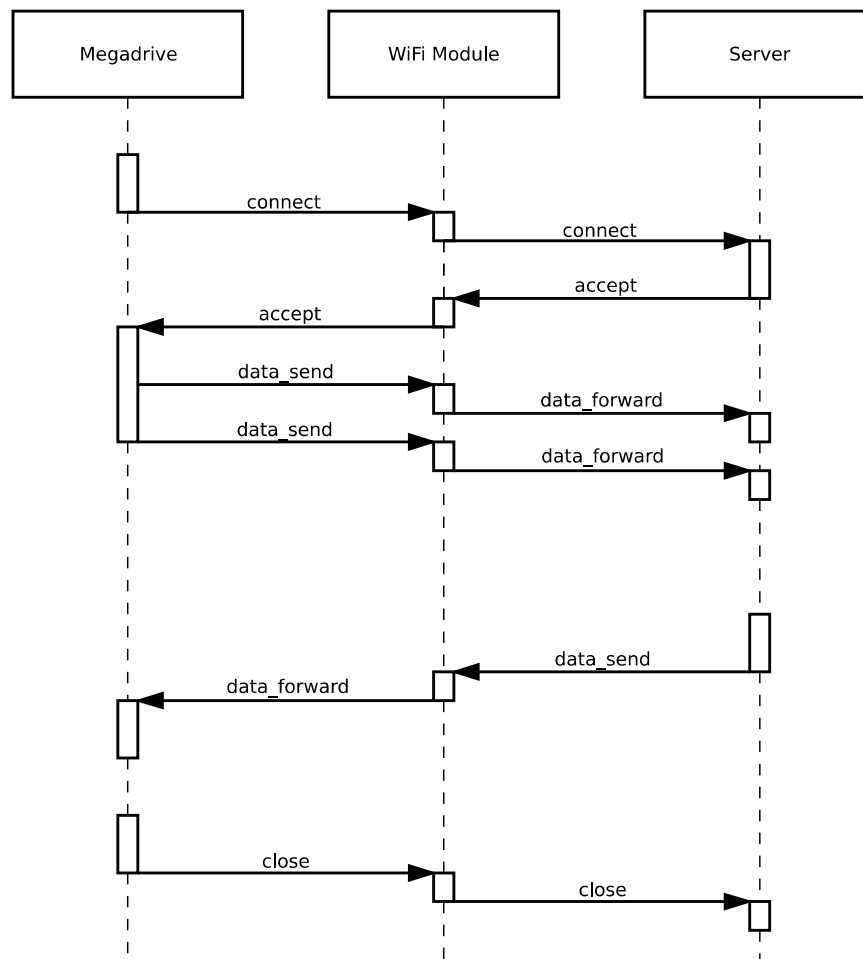
Figure 11: Typical TCP socket usage sequence

The first step when using TCP sockets, is to establish the connection with the server. Figure 11 shows how it is done when the *Megadrive* initiates the connection. Other alternative is having the *Megadrive* reserve a local TCP port and wait from incoming connections. When using UDP, the connection step is not performed.

The *WiFi Module* forwards the connection attempt to the server, and if the connection is successful (accepted), it sends the accept message to the *Megadrive*. Once the connection is accepted, both ends can start interchanging data. Figure 11 shows the client sending two data chunks, that are forwarded to the *Server* once received by the *WiFi module*. Some time later, the *Server* sends a data segment. This segment is received by the *WiFi module* and then forwarded to the *Megadrive*.

Finally the *Megadrive* closes the connection, and the close request is forwarded to the *Server* by the *WiFi module*. Any end can request the connection close, so on other scenarios it could be the *Server* who initiates the connection close. As happened with the connection stage, when using UDP,

there is no close stage.

TCP implements a flow control mechanism, so data cannot be sent to/from the *Server* until there is space on reception buffers (the TCP sliding window). A flow control must also be implemented between the *Megadrive* and the *WiFi Module*, using the UART RTS/CTS protocol, to avoid losing data. Again, on the other hand, UDP does not implement flow control mechanisms (neither guarantees data delivery nor correct order delivery).

Although it is not shown on Figure 11, several sockets can be simultaneously opened (unless using *transparent mode*). The data of each open socket must be sent using a different LSD channel. Therefore the maximum number of simultaneous sockets is limited to the available channels.

3.2.3 Using transparent mode

Transparent mode is a special mode of operation, used to further minimize protocol overhead over the serial line, maximizing throughput. When using *transparent mode*, LSD protocol framing is eliminated, at the cost of using only a single socket to communicate, and not being able to issue any other command to the *WiFi module* while this socket is opened.
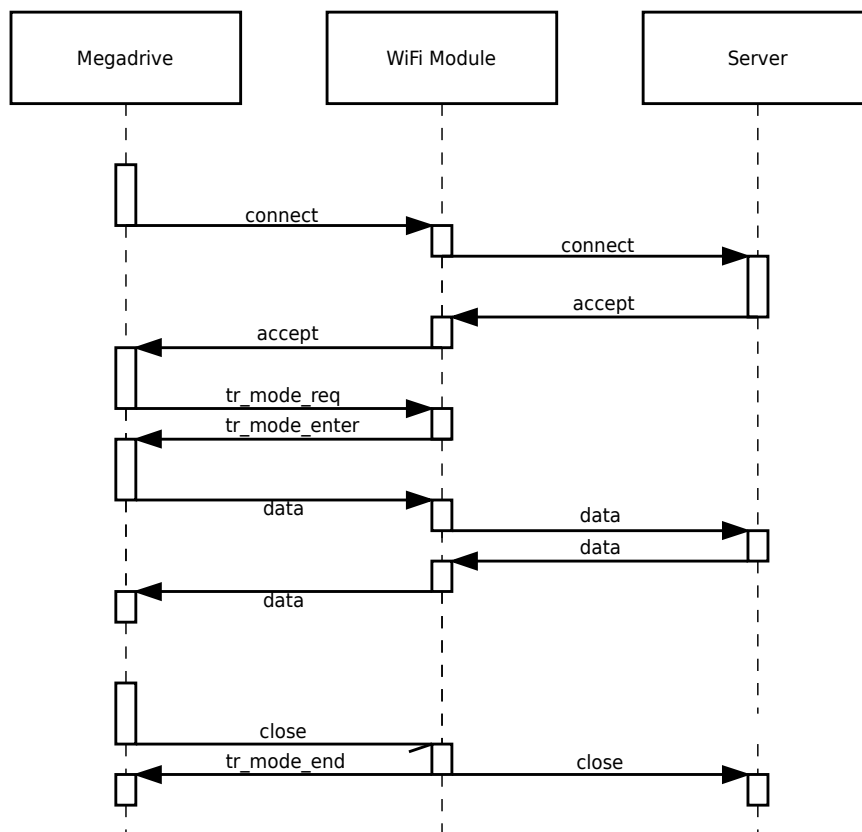
Figure 12: Transparent mode usage sequence

Figure 12 shows the typical usage of *transparent mode*. The *Megadrive* opens a single socket, and once the connection is established (if using TCP), it can request transparent mode. If the required conditions are met (only a single socket is open at the time of placing the request), a confirmation is sent to the *Megadrive* (tr_mode_enter) and both the *WiFi module* command interpreter and the LSD framing are disabled. Data sent by the *Megadrive* is forwarded to the

Server (over TCP/UPD/IP) as is, without any kind of framing/encapsulation. Data received by the *WiFi module* from the *Server* is also sent to the *Megadrive* the same way, without any kind of framing.

As there is neither framing nor command interpreter on the *WiFi module*, to end the connection, out of band signaling must be used. If the *Megadrive* wants to end the connection, as shown on Figure 12, it can request to close the socket using out of band signaling (activating a dedicated output line on the UART interface). When the *WiFi module* receives the close request, it closes the *Server* socket, and acknowledges the request, ending transparent mode, and thus reactivating the command interpreter and the LSD framing.

3.3 System state machine

MeGaWiFi firmware is controlled by a FSM (Finite State Machine) that handles the UART and its command interpreter, the wireless interface, the socket layer and internal system events. State machine design is shown in Figure 13.
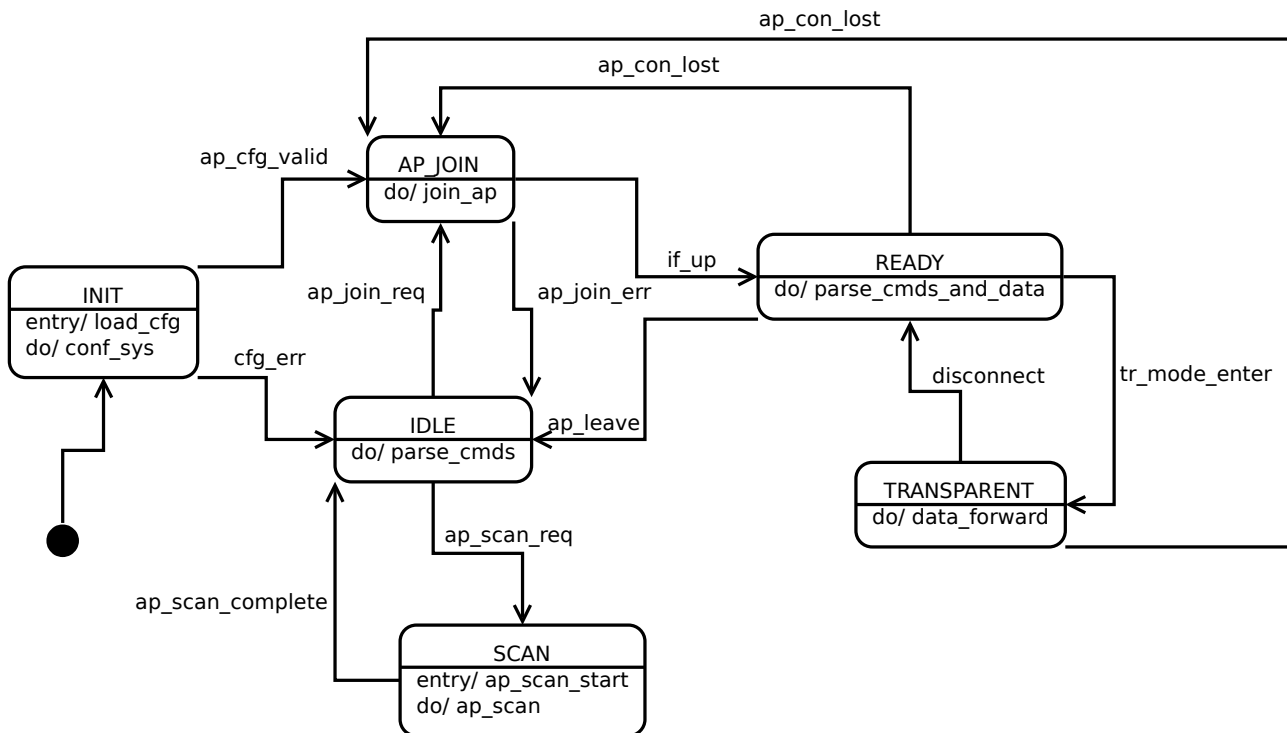
Figure 13: MeGaWiFi finite state machine

Events shown on Figure 13 are as follows:

- INIT: Initial state. The system configuration is loaded from flash memory, and system is configured, bringing up the UART and the wireless interface.

- IDLE: Idle state. The system is configured, but is not connected to an AP (Access Point). While in this state, system commands (sent over LSD channel 0) are parsed, but socket operations will fail. To end this state, an AP join request must be received with a valid AP configuration.

- SCAN: This state performs a scan for near access points. On entry the scan is started. Once scan is finished, the state is left.

- AP_JOIN: While in this state, the system tries to join specified AP. On success (AP joined and IPv4 subsystem properly configured), the system jumps to READY state.

- READY: Network is up, and the system is ready to operate. The FSM parses system commands on channel 0, is able to perform network socket operations and redirects data sent to channels other than 0.

- TRANSPARENT: This state can only be entered from READY, when *transparent mode* is requested, and a single socket is opened. While in this mode, command interpreter and LSD framing are disabled, and data is forwarded through the active socket. This state can only be left when the socket is closed (by the server, or using out of band request) or when a network error occurs (such as an AP disconnect).

The most important events parsed by the FSM are the following:

- ap_cfg_valid: Access point configuration loaded from Flash memory looks correct. Note: this event might not be modeled as an event, but as a check on INIT state.

- cfg_err: Configuration (including access point) is not correct. Note: this event might not be modeled as an event, but as a check on INIT state.

- ap_scan_req: An AP scan request has been received.

- ap_scan_complete: AP scan has been completed.

- ap_join_req: An AP join request has been received (and a valid AP configuration is stored).

- ap_join_err: Couldn't join the requested AP.

- if_up: System has joined requested AP, and IP layer configuration is complete (either statically specified on AP configuration, or received by DHCP).

- ap_leave: A request to detach from current AP has been received.

- ap_con_lost: Connection with AP has been lost.

- tr_mode_enter: A request to enter transparent mode has been received.

- disconnect: Socket connection has been terminated.

Please note the state machine briefly describes system behavior, and does not cover all the states and sub state machines required to fully implement the system (specially the socket related stuff).

3.4 Supported commands

As previously stated, MeGaWiFi firmware includes a command interpreter that receives and parses commands on LSD channel 0. Command frames consist of a command field (*cmd*), a data

length field (*len*) and an optional data payload (*data*), as shown on figure 14. Both the command and the data length fields take two bytes and are in *big endian* format. This frame format is used for both the command requests and the command responses.
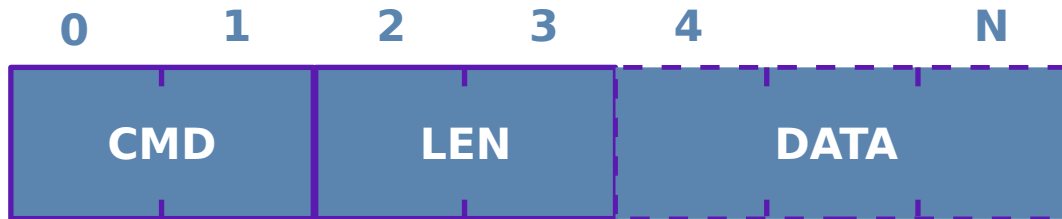

Figure 14: MeGaWiFi API command format

The command field contains a numeric identifier of the requested command or the result of the command execution (OK/ERROR). Supported command codes are shown on table 6, and explained below. For the ERROR command responses, an optional data payload may contain a text message related to the error.

3.4.1 VERSION

Requests the firmware version number and variant from the firmware running on the WiFi module. The version number is defined with two bytes (major version number and minor version number). The variant string contains a text string relative to the firmware variant. The "standard" firmware detailed on this document should always return the string "std" (standard).

- Payload length: 0.

- Reply:

    ○ OK: Firmware version obtained.

        ▪ Length: variable (from 2 to N bytes).

        ▪ Format: Major version number (1 byte) + minor version number (1 byte) + variant string (variable size, 3 bytes for the "standard" firmware variant).

    ○ ERROR: Could not obtain the firmware version number.

3.4.2 ECHO

Requests the host to echo the data contained on the command request. Used mainly for debugging purposes.

- Payload length: variable (the size of the message to echo).

- Format:

- Reply:

    ○ OK: Data echoed.

        ▪ Length: variable (should match the payload length of the ECHO request).

- Format: echoed message as sent on command request.
  - ERROR: Echo request failed.

Table 6: MegaWiFi API commands

Command	Code	Description
OK	0	Used on command replies to indicate the command was successful
VERSION	1	Request firmware version
ECHO	2	Request the host to echo command data. Used mainly for debugging
AP_SCAN	3	Start scanning access points
AP_CFG	4	Configure an access point
AP_CFG_GET	5	Get access point configuration
IP_CFG	6	Configure IPv4 network
IP_CFG_GET	7	Get IPv4 network configuration
AP_JOIN	8	Join an access point
AP_LEAVE	9	Leave a previously joined access point
TCP_CON	10	Create a TCP socket and try establishing a connection
TCP_BIND	11	Create a TCP socket and bind it to a port
TCP_ACCEPT	12	Accept an incoming TCP connection
TCP_STAT	13	Show TCP connection status
TCP_DISC	14	Disconnect a TCP connected socket
UDP_SET	15	Create a UDP socket and bind it to a port
UDP_STAT	16	Get UDP socket status
UDP_CLR	17	Clear (remove) a previously configured UDP socket
PING	18	Ping request
SNTP_CFG	19	Configure SNTP parameters
DATETIME	20	Get date and time
DT_SET	21	Set date and time
FLASH_WRITE	22	Write to WiFi module Flash
FLASH_READ	23	Read from WiFi module Flash
FLASH_ERASE	24	Erases a 4 KiB sector from the WiFi module flash
FLASH_ID	25	Obtains flash chip identifier codes of the WiFi module flash chip
ERROR	255	Used on command replies to indicate command was unsuccessful

3.4.3 AP_SCAN

Scans for access points. This command might take several seconds to complete.

- Payload length: 0

- Reply:

  ○ OK: AP scan complete.

    ■ Length: variable (depending on the number of APs found).

    ■ Format: A list of the received access points, filling the size indicated on the command reply. Each of the access points reported consists of the following fields:

      • Authentication mode (1 byte): type of authentication required to join the AP. Authentication codes are detailed on table 7.

      • Channel (1 byte): channel number used by the AP.

      • RSSI (1 byte): signal strength. Note this value is an 8-bit signed integer, usually negative.

      • SSID length (1 byte): length of the SSID field.

      • SSID (variable): SSID string, with the length indicated on the previous field. Maximum SSID length is 32 bytes. Note this string is not **null** terminated.

  ○ ERROR: Could not complete AP scan.

Table 7: Authentication modes

Code	Authentication mode
0	Open
1	WEP
2	WPA PSK
3	WPA2 PSK
4	WPA/WPA2 PSK
≥ 5	UNKNOWN

3.4.4 AP_CFG

Configures an access point. Up to 3 access point configurations are supported. These 3 configurations are linked to the ones specified by the IP_CFG command. Configurations set with this command are stored on the WiFi module Flash.

- Payload length: 97 bytes.

- Format:

  ○ Configuration number (1 byte): from 0 to 2.

  ○ SSID (32 bytes): SSID string, zero padded to fill 32 bytes.

○ Password (64 bytes): AP password, zero padded to fill 64 bytes.

- Reply:

    ○ OK: Access point successfully configured.

        ▪ Length: 0 bytes.

    ○ ERROR: Couldn't set access point configuration.

3.4.5 AP_CFG_GET

Obtain access point configuration. Read section 3.4.4 for more details.

- Payload length: 1 byte.

- Format:

    ○ Configuration number (1 byte): number of the configuration to obtain.

- Reply:

    ○ OK: Access point configuration successfully obtained.

        ▪ Length: 96 bytes.

        ▪ Format:

            • SSID (32 bytes): SSID string, zero padded to fill 32 bytes.

            • Password (64 bytes): AP password, zero padded to fill 64 bytes.

    ○ ERROR: Could not get specified access point configuration.

3.4.6 IP_CFG

IPv4 configuration command. Up to 3 configurations are supported, tied to the access point configurations set with AP_CFG command. If an IP_CFG is not set, or is set blank, DHCP will be used to get the IP configuration parameters. Configured IP parameters using this command are stored on the non volatile flash memory of the WiFi module.

- Payload length: 24 bytes.

- Format:

    ○ Configuration number (1 byte): from 0 to 2.

    ○ Reserved (3 bytes).

    ○ IP address (4 bytes): IPv4 address assigned to the WiFi interface, in binary format.

    ○ Net Mask (4 bytes): Subnet mask assigned to the WiFi interface, in binary format.

    ○ Gateway (4 bytes): IPv4 address of the Internet gateway, in binary format.

- ○ DNS1 (4 bytes): 1st domain name server IPv4 address in binary format.

- ○ DNS2 (4 bytes): 2nd domain name server IPv4 address in binary format.

- • Reply:

  - ○ OK: IPv4 configuration successfully set.

    - ▪ Length: 0 bytes.

  - ○ ERROR: Could not set specified IPv4 configuration.

3.4.7 IP_CFG_GET

Obtain specified IPv4 configuration. Read section 3.4.6 for more details.

- • Payload length: 1 byte.

- • Format:

  - ○ Configuration number (1 byte): from 0 to 2.

- • Reply:

  - ○ OK: Requested IPv4 configuration successfully obtained.

    - ▪ Length: 20 bytes

    - ▪ Format:

      - • IP address (4 bytes): IPv4 address assigned to the WiFi interface, in binary format.

      - • Net Mask (4 bytes): Subnet mask assigned to the WiFi interface, in binary format.

      - • Gateway (4 bytes): IPv4 address of the Internet gateway, in binary format.

      - • DNS1 (4 bytes): 1st domain name server IPv4 address in binary format.

      - • DNS2 (4 bytes): 2nd domain name server IPv4 address in binary format.

  - ○ ERROR: Could not get requested configuration.

3.4.8 AP_JOIN

Tries joining a previously configured access point.

- • Payload length: 1 byte.

- • Format:

  - ○ Configuration number (1 byte): from 0 to 2.

- • Reply:

- OK: Successfully joined requested AP.
  - Length: 0 bytes.
- ERROR: Could not join requested AP.

3.4.9 AP_LEAVE

Closes all opened sockets and leaves the currently joined access point.

- Payload length: 0 bytes.
- Reply:
  - OK: AP left.
    - Length: 0 bytes.
  - ERROR: Could not leave current AP.

3.4.10   TCP_CON

Creates a TCP socket and tries establishing a connection.

- Payload length: variable.
- Format:
  - Destination port (6 bytes): null terminated string, zero padded to fill 6 bytes.
  - Source port (6 bytes): null terminated string, zero padded to fill 6 bytes. If set to 0 or empty string, source port will automatically be assigned.
  - Channel (1 byte): Channel number to use for this socket by the LSD protocol line multiplexer. This value can be from 1 to (LSD_MAX_CH − 1), and can be used as a socket identifier.
  - Server name (variable): character string with the server name. Can be an IPv4 address or a resolvable domain name.
- Reply:
  - OK: Connection established.
    - Length: 0 bytes.
  - ERROR: Could not establish connection.

3.4.11   TCP_BIND

Creates a TCP socket and tries binding it to the specified port.

- Payload length: 7 bytes.
- Format:

- ○ Reserved (4 bytes): Set each of these bytes to 0.

- ○ Port (2 bytes): Port to which the socket will be bound.

- ○ Channel (1 byte): Channel number to use for this socket by the LSD protocol line multiplexer. This value can be from 1 to (LSD_MAX_CH − 1), and can be used as a socket identifier.

- • Reply:

  - ○ OK: Socket created and bound to requested port.

    - ▪ Length: 0 bytes.

  - ○ ERROR: Could not bind socket to requested port.

3.4.12   TCP_ACCEPT

Accepts an incoming TCP connection on a previously bound socket (with the command TCP_BIND).

- • Payload length: 1 byte.

- • Format:

  - ○ Channel (1 byte): Channel number associated with the socket that will accept the incoming connection.

- • Reply:

  - ○ OK: Connection accepted, socket is ready to send and receive data.

    - ▪ Length: 0 bytes.

  - ○ ERROR: Could not accept an incoming connection on requested socket.

3.4.13   TCP_STAT

Obtains the TCP status of the socket associated with the specified channel.

- • Payload length: 1 byte.

- • Format:

  - ○ Channel (1 byte): Channel number of the socket to poll.

- • Reply:

  - ○ OK: Socket status obtained.

    - ▪ Length: TBD!

    - ▪ Format: TBD!

  - ○ ERROR: Could not obtain socket status.

3.4.14   TCP_DISC

Disconnects a previously connected socket, and frees it. It also cancels and frees unconnected but bound sockets.

- Payload length: 1 byte.

- Format:

    ○ Channel (1 byte): Channel number of the socket to disconnect and free.

- Reply:

    ○ OK: Socket disconnected.

        ▪ Length: 0 bytes.

    ○ ERROR: Could not disconnect socket.

3.4.15   UDP_SET

Creates and configures an UDP socket, to send/receive datagrams to/from specified addresses and ports.

- Payload length: 24 bytes.

- Format:

    ○ Destination port (6 bytes): null terminated string, zero padded to fill 6 bytes.

    ○ Source port (6 bytes): null terminated string, zero padded to fill 6 bytes.

    ○ Channel (1 byte): Channel number to use for this socket by the LSD protocol line multiplexer. This value can be from 1 to (LSD_MAX_CH − 1), and can be used as a socket identifier.

    ○ Server name (variable): character string with the server name. Can be an IPv4 address or a resolvable domain name.

- Reply:

    ○ OK: UDP socket configured established.

        ▪ Length: 0 bytes.

    ○ ERROR: Could not create/configure UDP socket.

3.4.16   UDP_STAT

Obtains the UDP status of the socket associated with the specified channel.

- Payload length: 1 byte.

- Format:

◦ Channel (1 byte): Channel number of the socket to poll.

- Reply:

    ◦ OK: Socket status obtained.

        ▪ Length: TBD!

        ▪ Format: TBD!

    ◦ ERROR: Could not obtain socket status.

3.4.17   UDP_CLR

Clears and frees a previously set UDP socket.

- Payload length: 1 byte.

- Format:

    ◦ Channel (1 byte): Channel number of the socket to free.

- Reply:

    ◦ OK: Socket freed.

        ▪ Length: 0 bytes.

    ◦ ERROR: Could not free socket.

3.4.18   PING

TBD!

3.4.19   SNTP_CFG

Configures SNTP time synchronization client service. This configuration is stored on the non volatile flash memory of the WiFi module.

- Payload length: variable

- Format:

    ◦ Update delay (2 bytes): The number of seconds between consecutive time update requests to the NTP servers. Minimum value is 15 seconds.

    ◦ Timezone (1 byte): Timezone information used to adjust NTP time (signed 8-bit integer, from -11 to 13).

    ◦ NTP servers (variable): The IP addresses or resolvable domain names of up to three NTP servers. Each server is null terminated. The end of the list is marked by two consecutive null characters

- Reply:

- OK: SNTP configuration successfully applied.

  - Length: 0 bytes.

- ERROR: Could not set requested SNTP configuration.

3.4.20 DATETIME

Obtains the date and time keep by the WiFi module. **Note**: the value returned will be wrong until at least one NTP time update is received or the date and time has been set using DT_SET command. **Warning**: date and time is automatically adjusted by the SNTP service, and is not guaranteed to be monotone increasing. If you need the date and time to be monotone increasing, do not set SNTP servers and manually set/correct the date and time. Also be warned that the date and time is not very accurate unless constantly adjusted by the SNTP service.

- Payload length: 0 bytes

- Reply:

  - OK: date and time successfully obtained.

    - Length: variable.

    - Format:

      - Secs (8 bytes): The number of seconds since the Epoch.

      - Date Time (variable) : The date and time in text format. E.g.: "Thu Mar 3 12:26:51 2016".

3.4.21 DT_SET

Sets the date and time.

- Payload length: 8 bytes.

- Format:

  - Secs (8 bytes): The number of seconds since the Epoch to set the date and time to.

- Reply:

  - OK: date and time set.

    - Length: 0 bytes.

  - ERROR: Could not set date and time.

3.4.22 FLASH_WRITE

Writes data to the SPI flash chip inside the ESP8266 WiFi module. These modules usually contain a 32 megabit SPI flash chip, and only 4 megabits are used for the firmware. So the remaining 28 megabits can be used by the application running on the Megadrive console.

Note: Firmware memory flash range is protected and cannot be accessed by flash writes.

Warning: For the write operations to succeed, the memory locations that are written to, must be erased (readed as 0xFF). Writing on non erased memory locations will most likely cause data corruption. Use the FLASH_ERASE command to erase flash memory sectors.

- Payload length: variable.

- Format:

  ○ Address (4 bytes): the address of the flash chip that will be written to. Addresses start at 0x00000000 and for the default 4 MiB chips, go up to 0x37FFFF.

  ○ Data: Data payload.

- Reply:

  ○ OK: data written to specified address.

    ▪ Length: 0 bytes.

  ○ ERROR: could not write data to specified location.

3.4.23   FLASH_READ

Reads data from the SPI flash chip inside the ESP8266 WiFi module. These modules usually contain a 32 megabit SPI flash chip, and only 8 megabits are used for the firmware. So the remaining 24 megabits can be used by the application running on the Megadrive console.

Note: Firmware memory flash range is protected and cannot be accessed by flash reads.

- Payload length: 6 bytes.

- Format:

  ○ Address (4 bytes): the address of the flash chip that will be read from. Addresses start at 0x000000 and for the default 4 MiB chips, go up to 0x37FFFF.

  ○ Length (2 bytes): the number of bytes to read from the specified address.

- Reply:

  ○ OK: data successfully read.

    ▪ Length: variable (matching the one specified on the request).

    ▪ Format:

      • Data: The data read from the flash chip, with length matching the request.

  ○ ERROR: could not read requested data from flash chip.

3.4.24   FLASH_ERASE

Erases a 4 KiB (32 kilobit) sector from the flash chip embedded on the ESP8266 WiFi module.

Erased memory is read as 0xFF, and can be programmed by FLASH_WRITE commands.

Warning: single bytes cannot be erased, a complete sector erase must be performed. Erasing a flash sector destroys its contents.

- Payload length: 2 bytes.

- Format

    ○ Sector number: number of the sector to erase. Sector number corresponding to an address can be obtained by shifting the address to the right 12 times (or dividing it by 2^{12}).

- Reply:

    ○ OK: Sector properly erased.

        ■ Length: 0 bytes.

    ○ ERROR: Could not erase requested sector.

3.4.25   FLASH_ID

Obtains the identifier codes of the flash chip embedded in the ESP8266 WiFi module. These codes can be used to determine the manufacturer and size of the chip.

- Payload length: 0 bytes.

- Reply:

    ○ OK: flash chip codes obtained.

        ■ Length: 4 bytes.

        ■ Format:

            • ManId (1 byte): manufacturer identifier.

            • DevId (2 bytes): device identifier

    ○ ERROR: could not read flash chip codes.

4 MeGaWiFi API for SEGA MegaDrive/Genesis

TBD!

5 Annex I. Obtaining sources

MegaWiFi source code for all its components (programmer firmware, command-line interface flash tool, WiFi module firmware, console API) is free. You can browse it, along with some more information and instructions about how to build the binaries, at GitHub:

https://github.com/doragasu/mw

Contributions are welcome. Please use the GitHub repositories to report bugs and send pull requests with corrections or new features you wish to be added to MegaWiFi.

6 Annex II. Programmer firmware upgrade

MeGaWiFi Programmer contains a microcontroller running a specially programmed firmware. The firmware inside the microcontroller is split in two programs:

- **mdma-bl**: a DFU bootloader used only to upgrade *mdma-fw*.

- **mdma-fw**: The program used to implement all the required functions of the programmer (USB communications with *mdma-cli* client, ROM programming on the cartridge flash memory, etc.).

On normal operation, *mdma-fw* program is started. The bootloader (*mdma-bl*) is only executed to upgrade *mdma-fw*. The upgrade process consists of two parts: first DFU bootloader mode must be entered, and then a DFU utility must be used to program the new firmware.

6.1 Requirements

To upgrade the firmware, you need the following items:

- The firmware blob you want to flash, provided as an ".hex" file (e.g. *mdma-fw.hex*).

- A MeGaWiFi programmer, already programmed with a working *mdma-bl* program. Please note *mdma-fw* must also be flashed and must be at least partially working (as *mdma-fw* is in charge of invoking *mdma-bl*).

- A PC running a suitable DFU utility (such as Atmel *FLIP for Windows* or *dfu-programmer*).

6.2 Entering DFU bootloader mode

To start the bootloader do one of the following actions:

- If connected, unplug the programmer from the host PC. Then press and hold SW1 pushbutton while connecting the USB cable to the host PC. Release SW1 once the programmer is connected to the PC.

- While mdma-fw is running and in idle state, use mdma-cli to enter bootloader mode (e.g. invoke `mdma --bootloader` on the command line).

If done correctly, the microcontroller will enter DFU bootloader mode, and will wait for DFU commands. To signal this state, LEDs D1 and D2 will start blinking alternatively.

6.3 Updating the programmer firmware

Once in DFU bootloader mode (with LEDs D1 and D2 blinking alternatively), you have to use a DFU utility to flash the new firmware. Below will be explained the process needed to do it using *dfu-programmer* utility under *GNU/Linux* OS., but alternatively you can use other DFU utilities, such as Atmel *FLIP for Windows*.

Once MeGaWiFi programmer is in DFU bootloader mode, to program *mdma-fw.hex* file using *dfu-programmer* utility, open a shell and do the following steps (only the commands following '$' character must be typed at the prompt):

```
$ dfu-programmer at90usb646 erase
Checking memory from 0x0 to 0xDFFF...  Not blank at 0x1.
Erasing flash...  Success
$ dfu-programmer at90usb646 flash mdma-fw.hex
Checking memory from 0x0 to 0x15FF...  Empty.
0%                          100%  Programming 0x1600 bytes...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>]  Success
0%                          100%  Reading 0xE000 bytes...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>]  Success
Validating...  Success
0x1600 bytes written into 0xE000 bytes memory (9.82%).
$ dfu-programmer at90usb646 reset
```

The first command erases the already burned firmware. The second command programs the new firmware. Finally, the third command reboots the programmer, that will run the upgraded firmware.

Alternatively, if working on the firmware project tree, and if LUFA library build environment has been properly configured, you can flash the firmware by invoking the command:

```
$ make dfu
```

> ⚠ **WARNING**: Make sure the programmed mdma-fw is at least partially working and is able to enter DFU bootloader mode. Otherwise you will not be able to upgrade the firmware anymore unless using an external tool (such as an ISP programmer).

> ⚠ **WARNING**: Make sure the firmware you program is designed specially for the MeGaWiFi Programmer board. Otherwise the board and/or cartridge can be damaged!

6.4 Fixing permissions

For these commands to work, you must have proper permissions. Otherwise read/write operations will fail. One way of granting these permissions to select users, is to add these users to a common group, and allow the group to read and write to the programmer interfaces. For example create (if it does not already exist) the group uucp. Then create the file /etc/udev/rules.d/99-mega-prog.rules with the following contents:

```
ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="2ff9", GROUP="uucp", MODE="0660"
ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="206c", GROUP="uucp", MODE="0660"
```

This ensures members of the group uucp can access the programmer both in bootloader and application mode.

7 Annex III. Batch ROM writing

The programmer includes a pushbutton that is readable using *mdma-cli* application. This allows to perform batch cartridge flashing by using a program or script that interfaces with *mdma-cli*. The following sample script can be used for this task on a Unix system:
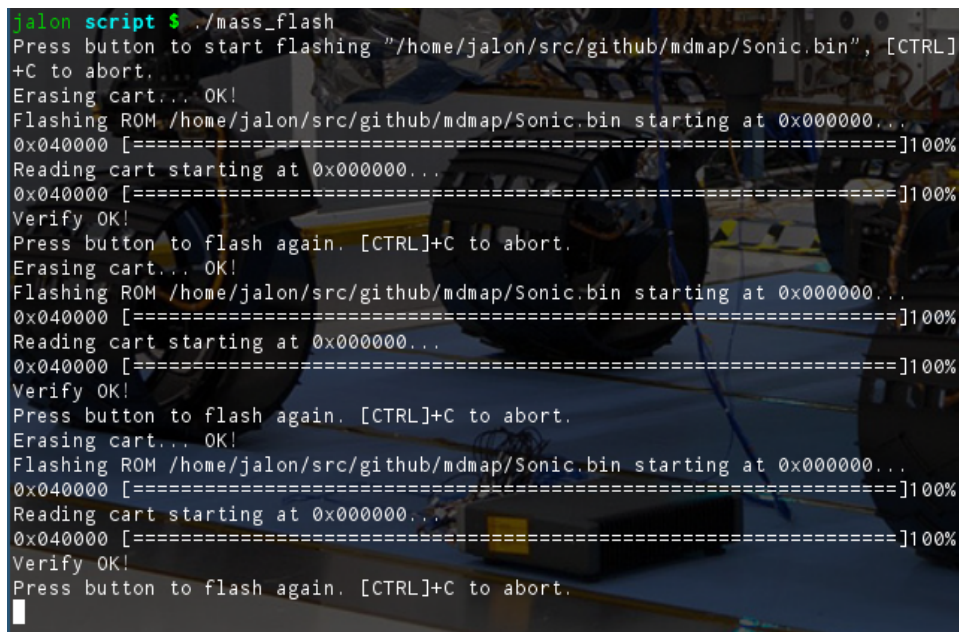
```sh
#!/bin/sh
FLASH=$HOME/src/github/mdmap/Sonic.bin
MDMA=$HOME/src/github/mdmap/mdmap

echo Press button to start flashing \"$FLASH\", [CTRL]+C to abort.

# Clear pending pushbutton events (if any)
$MDMA -p

# Flashing infinite loop. Detect a pushbutton event and then start flashing
while true
do
  $MDMA -p
  if [ $? -ge 2 ]
  then
        $MDMA -Vef "$FLASH"
        # Clear events occurred during flashing
        echo Press button to flash again. [CTRL]+C to abort.
        $MDMA -p
  fi
  # Sleep 250 ms between iterations
  sleep 0.25
done
```

An invocation of this script is shown on Figure 15.

Figure 15: mass-flash script invocation.