

# MegaWiFi

---

## WiFi connectivity for the SEGA Genesis/Megadrive console

Designed and implemented for 1985alternative  
by doragasu (2015 – 2020)

# Index

1	Introduction.....	9
1.1	Capabilities.....	10
1.1.1	WiFi and network connectivity.....	10
1.1.2	Date and time synchronization.....	11
1.1.3	Non-volatile read/write storage.....	11
1.1.4	Hardware random number generator.....	11
1.2	Navigating this document.....	11
2	Cartridge programmer software.....	13
2.1	MegaDrive Memory Administration Protocol (MDMAP).....	13
2.1.1	USB requirements.....	14
2.1.2	Command specification.....	15
	MDMA_MANID_GET.....	15
	MDMA_DEVID_GET.....	15
	MDMA_READ.....	16
	MDMA_CART_ERASE.....	16
	MDMA_SECT_ERASE.....	17
	MDMA_WRITE.....	17
	MDMA_MAN_CTRL.....	18
	MDMA_BOOTLOADER.....	19
	MDMA_BUTTON_GET.....	19
	MDMA_WIFI_CMD.....	20
	MDMA_WIFI_CMD_LONG.....	20
	MDMA_WIFI_CTRL.....	21
	MDMA_RANGE_ERASE.....	21
2.2	Host PC application.....	22
2.2.1	Qt GUI.....	22
2.2.2	Command line interface.....	23
3	MegaWiFi firmware for ESP8266.....	26

3.1 Use cases.....	26
3.1.1 Access point management.....	27
3.1.2 TCP sockets.....	28
3.1.3 UDP sockets.....	29
3.1.4 Misc functions.....	30
3.2 Communications flow.....	30
3.2.1 LSD Protocol.....	32
3.2.2 Principle of operation.....	33
3.2.3 Using transparent mode.....	35
3.3 System state machine.....	36
3.4 Supported commands.....	37
3.4.1 VERSION.....	38
3.4.2 ECHO.....	38
3.4.3 AP_SCAN.....	40
3.4.4 AP_CFG.....	41
3.4.5 AP_CFG_GET.....	41
3.4.6 IP_CURRENT.....	42
3.4.7 IP_CFG.....	42
3.4.8 IP_CFG_GET.....	43
3.4.9 DEF_AP_CFG.....	43
3.4.10 DEF_AP_CFG_GET.....	44
3.4.11 AP_JOIN.....	44
3.4.12 AP_LEAVE.....	44
3.4.13 TCP_CON.....	44
3.4.14 TCP_BIND.....	45
3.4.15 CLOSE.....	45
3.4.16 UDP_SET.....	46
3.4.17 SOCK_STAT.....	46
3.4.18 PING.....	47

3.4.19	<i>SNTP_CFG</i> .....	47
3.4.20	<i>SNTP_CFG_GET</i> .....	47
3.4.21	<i>DATETIME</i> .....	48
3.4.22	<i>DT_SET</i> .....	48
3.4.23	<i>FLASH_WRITE</i> .....	49
3.4.24	<i>FLASH_READ</i> .....	49
3.4.25	<i>FLASH_ERASE</i> .....	50
3.4.26	<i>FLASH_ID</i> .....	50
3.4.27	<i>SYS_STAT</i> .....	50
3.4.28	<i>DEF_CFG_SET</i> .....	52
3.4.29	<i>HRNG_GET</i> .....	52
3.4.30	<i>BSSID_GET</i> .....	53
3.4.31	<i>GAMERTAG_SET</i> .....	53
3.4.32	<i>GAMERTAG_GET</i> .....	54
3.4.33	<i>LOG</i> .....	54
3.4.34	<i>FACTORY_RESET</i> .....	54
3.4.35	<i>SLEEP</i> .....	55
3.4.36	<i>HTTP_URL_SET</i> .....	55
3.4.37	<i>HTTP_METHOD_SET</i> .....	55
3.4.38	<i>HTTP_CERT_QUERY</i> .....	56
3.4.39	<i>HTTP_CERT_SET</i> .....	56
3.4.40	<i>HTTP_HDR_ADD</i> .....	57
3.4.41	<i>HTTP_HDR_DEL</i> .....	57
3.4.42	<i>HTTP_OPEN</i> .....	57
3.4.43	<i>HTTP_FINISH</i> .....	58
3.4.44	<i>HTTP_CLEANUP</i> .....	58
3.4.45	<i>SERVER_URL_GET</i> .....	58
3.4.46	<i>SERVER_URL_SET</i> .....	59
4	MegaWiFi API for SEGA MegaDrive/Genesis.....	60

4.1	Introduction.....	60
4.2	API documentation.....	60
4.3	Building.....	60
4.4	Overview.....	60
4.4.1	Loop module.....	60
4.4.2	Mpool module.....	63
4.4.3	Megawifi module.....	64
4.5	Putting all together.....	65
4.5.1	Connection configuration.....	65
4.5.2	Program initialization.....	65
4.5.3	Associating to an AP.....	67
4.5.4	Connecting to a TCP server.....	68
4.5.5	Creating a TCP server socket.....	68
4.5.6	Sending data.....	69
4.5.7	Receiving data.....	69
4.5.8	Performing an HTTP/HTTPS request.....	70
4.5.9	Getting the date and time.....	73
4.5.10	Setting and getting gamertag information.....	74
4.5.11	Reading the module BSSIDs.....	74
4.5.12	Reading and writing to non-volatile Flash.....	75
5	Annex I. Obtaining sources.....	76
6	Annex II. Programmer firmware upgrade.....	77
6.1	Requirements.....	77
6.2	Entering DFU bootloader mode.....	77
6.3	Updating the programmer firmware.....	77
6.4	Fixing permissions.....	78
7	Annex III. Batch ROM writing.....	79



## Figure Index

Figure 1: MegaWiFi programmer.....	9
Figure 2: MegaWiFi cartridge.....	9
Figure 3: MegaWiFi blocks.....	9
Figure 4: MDMA_WRITE command.....	13
Figure 5: Access point management.....	26
Figure 6: TCP sockets.....	27
Figure 7: UDP sockets.....	28
Figure 8: Misc functions.....	29
Figure 9: MegaWiFi Communications flow.....	30
Figure 10: LSD PDU format.....	31
Figure 11: Typical TCP socket usage sequence.....	33
Figure 12: Transparent mode usage sequence.....	34
Figure 13: MegaWiFi finite state machine.....	35
Figure 14: MegaWiFi API command format.....	37
Figure 15: System status flags.....	51
Figure 16: mass-flash script invocation.....	78

## Table Index

Table 1: Supported MDMAP commands.....	12
Table 2: USB interface details.....	13
Table 3: Manual GPIO control command format.....	17
Table 4: Supported WiFi module control commands.....	20
Table 5: Command line interface options.....	23
Table 6: MegaWiFi API commands.....	38
Table 7: Authentication modes.....	40



# 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.

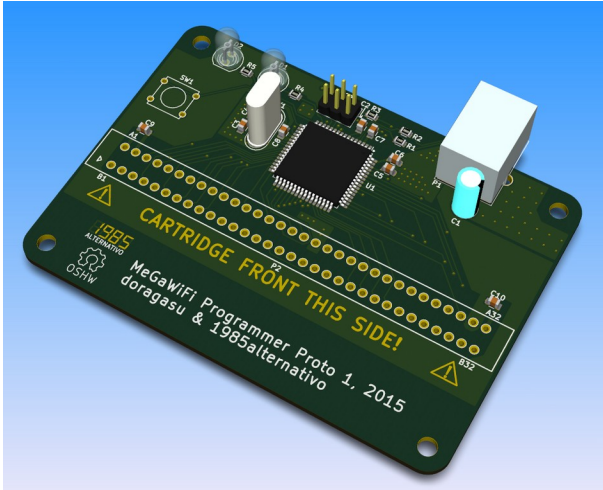


Figure 1: MegaWiFi programmer

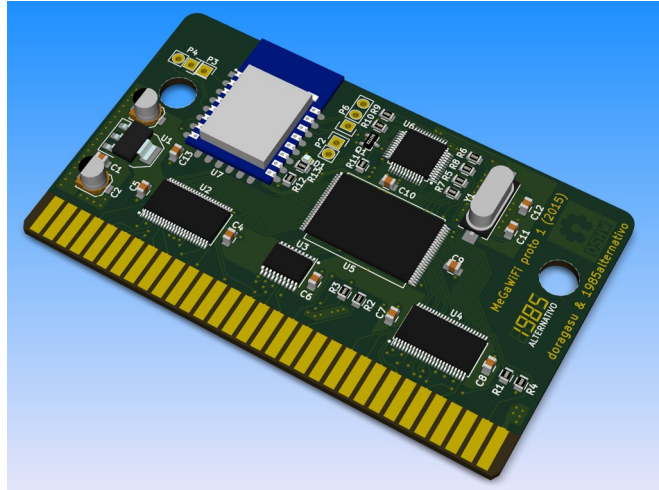


Figure 2: MegaWiFi cartridge

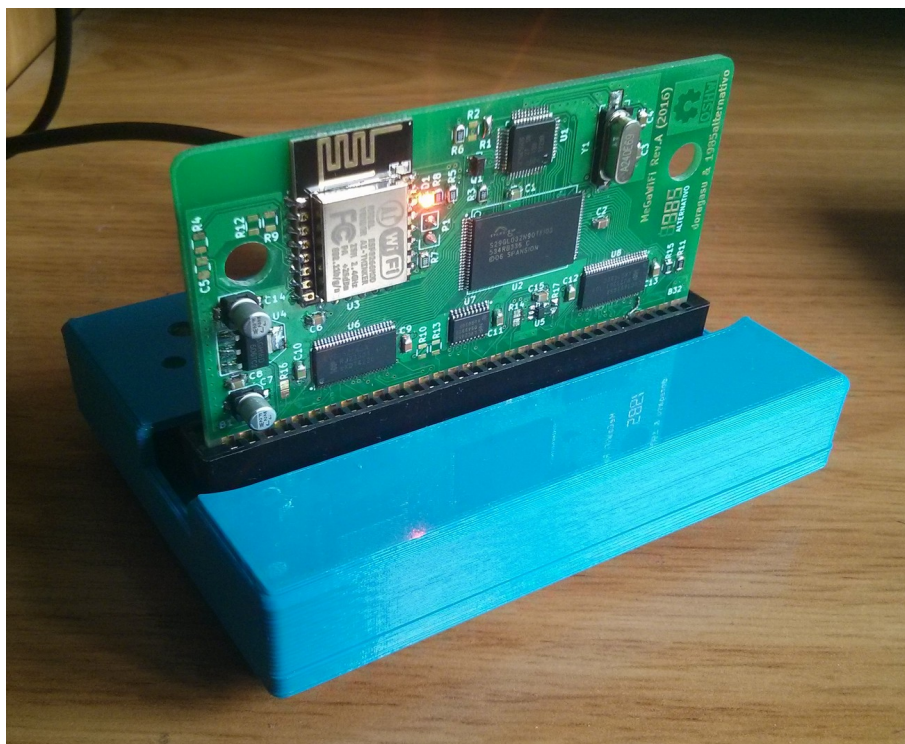


Figure 3: MegaWiFi cartridge inserted in the programmer

The cartridge consists of the blocks shown on Figure 4.

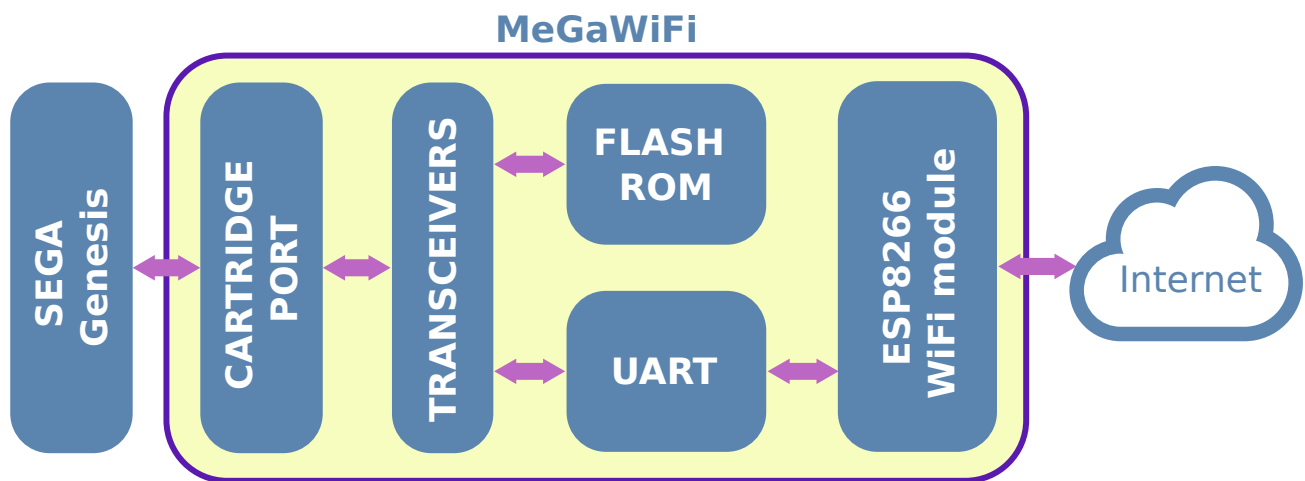


Figure 4: MegaWiFi blocks

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.

## 1.1 Capabilities

MegaWiFi cartridges contain a WiFi module with a built-in 32-bit CPU running at 80 MHz. A default firmware runs on this CPU providing the Genesis/Megadrive console with network capabilities and some more features. The firmware is Free Software, so it can be customized, adding additional features (like using the CPU as numeric co-processor for trigonometric computations) when needed. Unfortunately, as will be discussed below, bandwidth between the WiFi module and the console CPU is limited and requires using polled mode to reach maximum speed, greatly restricting the applications this module can be used for.

The following subsections detail the capabilities provided by the default firmware.

### *1.1.1 WiFi and network connectivity*

The main purpose of the cartridge is to add to the console WiFi connectivity, being able to connect to the Internet. Using MegaWiFi, the console can create TCP and UDP client and server sockets, to send and receive data to/from any host connected to the Internet through the IPv4 network protocol. MegaWiFi supports the operation of several simultaneous sockets.

WiFi radio is b/g/n compliant and supports standard authentication/encryption algorithms (WEP/WPA/WPA2). WiFi scanning is implemented and allows to obtain the SSID, channel, signal strength and authentication type of nearer Access Points. IP and DNS configuration can be obtained by the console automatically (by using the DHCP client built inside the module) or configured manually.

Theoretical maximum bit rate is 1.5 Mbps (limited by the serial connection between the WiFi module and the console CPU), but as sending/receiving is done using polling, this bit rate is only achievable when using all the available CPU (unless clever multiplexing techniques are used). If CPU usage is a concern (e.g. during an action game), bit rate may be limited to just around 16 bytes per frame (9600 bps for a 60 Hz system) without incurring on a CPU penalty. This might be enough for simple online games, but for more complex games, special techniques (e.g. using HINT scanline interrupts to receive data, or using the loop module) must be used to achieve higher bit rates without wasting too much CPU time.

### *1.1.2 Date and time synchronization*

MegaWiFi implements an SNTP time synchronization daemon. If properly configured, when the module connects to the Internet, it synchronizes its internal date and time. Since date and time is synchronized, the console can request the date and time, that will be retrieved from MegaWiFi internal clock. Module internal clock is low precision, so date and time is re-synchronized at regular intervals as long as the module is connected to the Internet.

There is however an important limitation to the module internal clock: date and time is not kept when the console is powered off, so to reliably use the clock feature, it is necessary to have Internet connectivity, for the synchronization mechanism to set the time.

Due to the relatively low precision of the clock and the background synchronization process, the clock is not guaranteed to be monotonic. This should not be a concern for most applications, but it must be taken into account if “going backwards in time” can cause problems.

### *1.1.3 Non-volatile read/write storage*

Along with up to 32 megabits (4 MiB) of read-only NOR flash memory, the WiFi module inside MegaWiFi cartridges contains 32 megabits of additional SPI read/write flash memory. The WiFi module firmware requires 8 megabits of this memory to work, so the remaining 24 megabits (3 MiB) can be used by the user program. This memory is non-volatile and read/write, so it can be used both to hold static data (that does not fit inside the main 32 megabits of memory) and also dynamic data (savegames, DLC contents, etc.).

Due to these 24 megabits of additional memory been located inside the WiFi module (that is connected to the console bus through the UART), there is a limitation on the maximum read/write bandwidth: 1.5 Mbps max. when using polled mode (using 100% CPU while reading/writing). Taking this limitation into account, it is recommended to only read/write data from/to this additional storage when action is not displayed on the screen (e.g. to load level data before starting it).

#### *1.1.4 Hardware random number generator*

The WiFi module contains a hardware random number generator. The speed of this module is again limited by the bandwidth of the connection between the console CPU and the WiFi module (1.5 Mbps in polled mode).

## 1.2 Navigating this document

This section deals with the MegaWiFi introduction.

Section 2 deals with the cartridge programmer, its firmware and the client software.

Section 3 details the firmware developed for the WiFi module inside the cartridge, and documents how to send and receive data, along with all the available commands the console can issue to the WiFi module.

Section 4 documents de MegaWiFi API that can be used to easily create Megadrive programs using MegaWiFi capabilities.

Annex I explains how to obtain MegaWiFi related sources.

Annex II shows how to upgrade the MegaWiFi programmer firmware.

Annex III explains how to write ROMs in batches to carts, using a MegaWiFi programmer.

After reading chapter 1, programmers wanting to code for the console using MegaWiFi, can skip chapter 2, quickly browse the first section of chapter 3, and jump directly to chapter 4, where the APIs they will be used are documented.

## 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.
13	MDMA_RANGE_ERASE	Erase a memory range of the flash chip

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 5, 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.

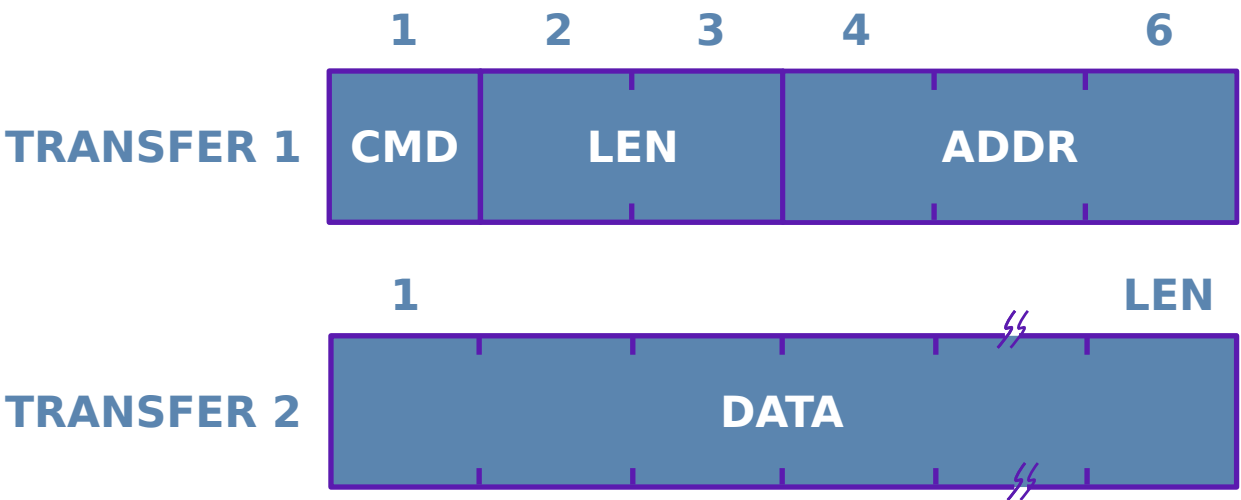


Figure 5: MDMA\_WRITE command

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 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 <sup>1</sup>
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:

---

<sup>1</sup> 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 of 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 transfer 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.

- 1<sup>st</sup> transfer size: 6 bytes.
- 1<sup>st</sup> transfer format: MDMA\_WRITE (1 byte) + *wLen* (2 bytes) + *addr* (3 bytes).
- 2<sup>nd</sup> transfer size: 2 to 65534 bytes (1 to 32767 words).
- 2<sup>nd</sup> 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.

**NOTE:** this for safety reasons, this command is not implemented, but documentation is kept for reference.

- 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/ $\overline{W}$ :	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 attribute: 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* or *Atmel Flip* 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 not 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.
- 1<sup>st</sup> transfer format: MDMA\_WIFI\_CMD (1 byte) + command length (2 bytes) + padding (1 byte).
- 2<sup>nd</sup> 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 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.

## **MDMA\_RANGE\_ERASE**

Erases a memory range from the Flash chip.

- Size: 6 bytes.
- Format: MDMA\_RANGE\_ERASE (1 byte) + *addr* (3 bytes) + *wLen* (2 bytes).
- Reply:
  - MDMA\_OK: Erase operation completed successfully.
    - Size: 1 byte.
    - Format: MDMA\_OK (1 byte).
  - MDMA\_ERR: Could not perform read operation.
    - Size: 1 byte.

- Format: MDMA\_ERR (1 byte).

## 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 supports both a Qt based graphical user interface, and a command line interface, to be able to integrate it with different toolchains.

The application is cross-platform and has been tested both on Windows 7, Windows 10 and several GNU/Linux distributions. It should also work on macos, but as I do not own a Mac, I cannot build and test it.

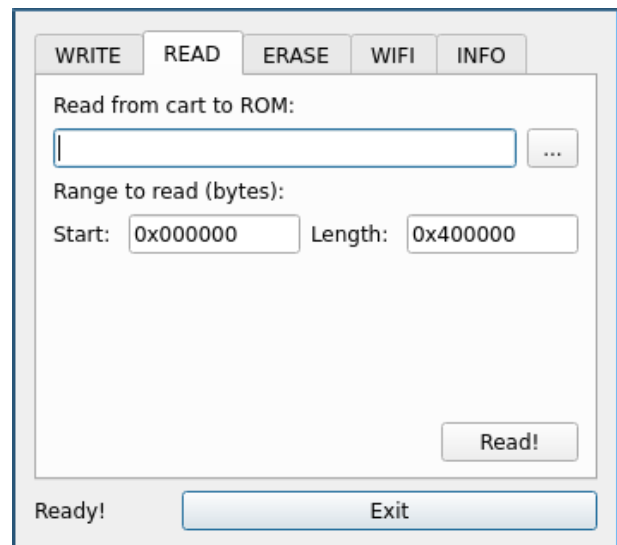
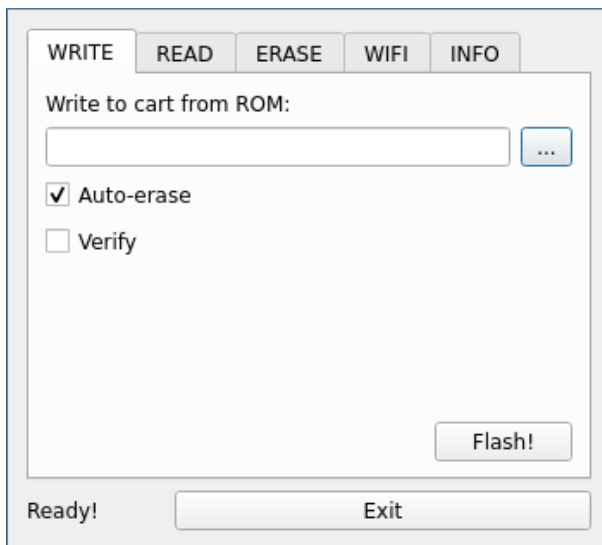
### 2.2.1 Qt GUI

The Qt based graphical interface allows managing the cartridge memory in an easy way. To start the Qt GUI, the application must be invoked with the `-Q` switch:

```
$ mdma -Q
```

The GUI consists of 5 tabs, each one used for a different operation with the cart:

- **WRITE**: allows writing ROMs to the cartridge.
- **READ**: allows reading the contents already burned to the cartridge.
- **ERASE**: allows erasing the flash memory of the cartridge. Note this is usually not needed, since the **WRITE** tab has an option to automatically erase the flash before writing.
- **WIFI**: allows writing firmware blobs to the WiFi chip. Currently supported only by the command-line interface.
- **INFO**: shows programmer and cartridge information.



WRITE

READ

ERASE

WIFI

INFO

☐ Full erase
 Range to erase (bytes):  
 Start:  Length: 

Erase!

Ready!

Exit

WRITE

READ

ERASE

WIFI

INFO

MDMA Version:  
  
 Programmer version:  
  
 Flash manufacturer ID:  
  
 Flash device IDs:  
  
 MegaDrive Memory Administration,  
 by Migue and doragasu, 2017
 

Bootloader mode

Ready!

Exit

### 2.2.2 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 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
--qt-gui, -Q	N/A	Starts the Qt GUI, as described in subsection 2.2.1
--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.
--range-erase, -A	R - Range	Erases the flash memory region containing the specified range.
--auto-erase, -a	N/A	Auto-erase before flash (use it with flash command).
--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 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).
- *Range*: Specifies a memory range, with the start address and length, separated by the colon (:) character, as follows:

```
range_start_address:range_length
```

- *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 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* invocation. 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 -Vaf rom_file:0x100000:32768` → Erases 32 KiB (or more) at address 0x100000, flashes 32 KiB of *rom\_file* to address 0x100000, and verifies the flash 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

**NOTE:** as of version 1.0 of the firmware, *transparent mode* is not supported.

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

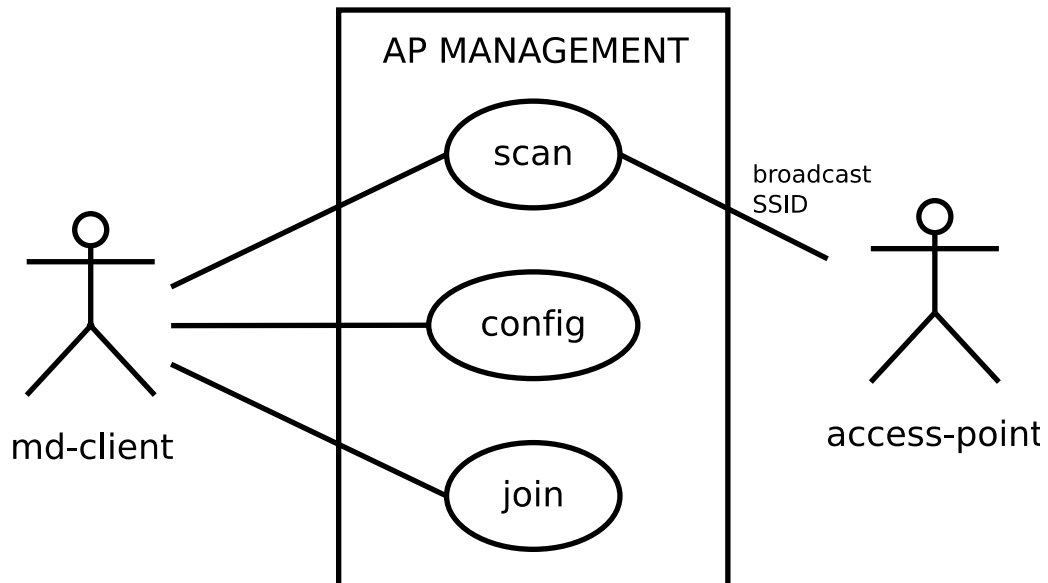


Figure 6: Access point management

- 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* (associate to) an AP.

### 3.1.2 TCP sockets

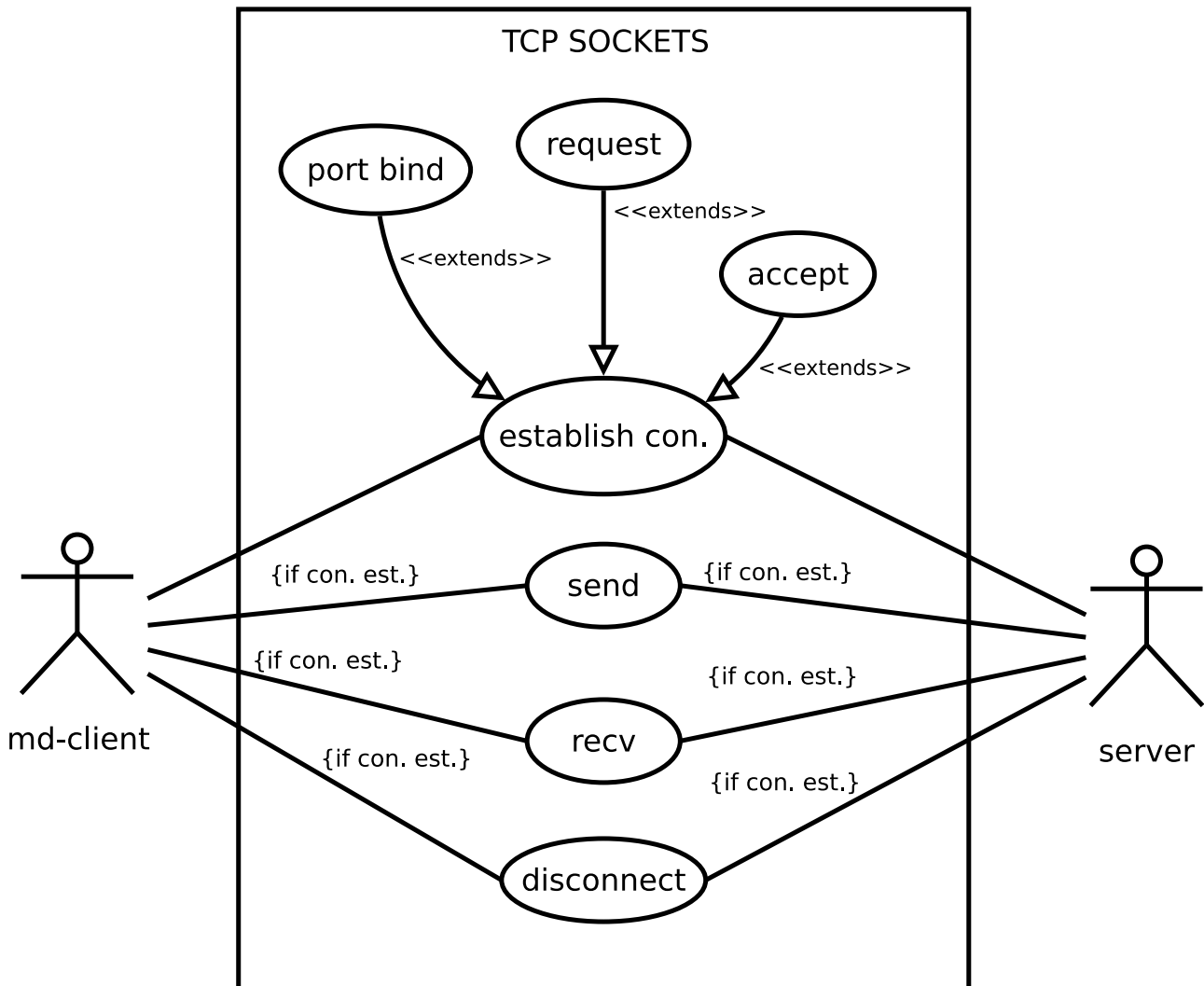


Figure 7: TCP sockets

- 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

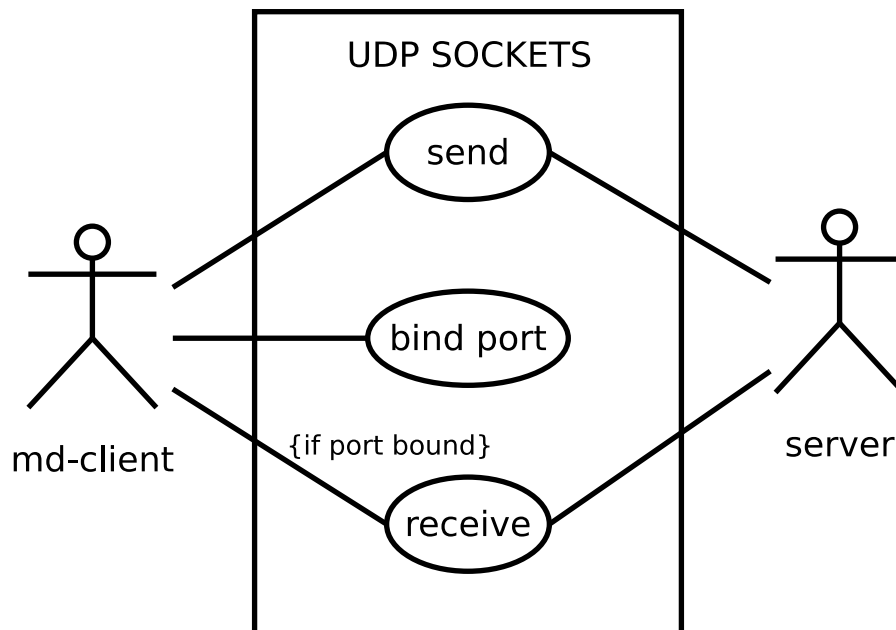


Figure 8: UDP sockets

- 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

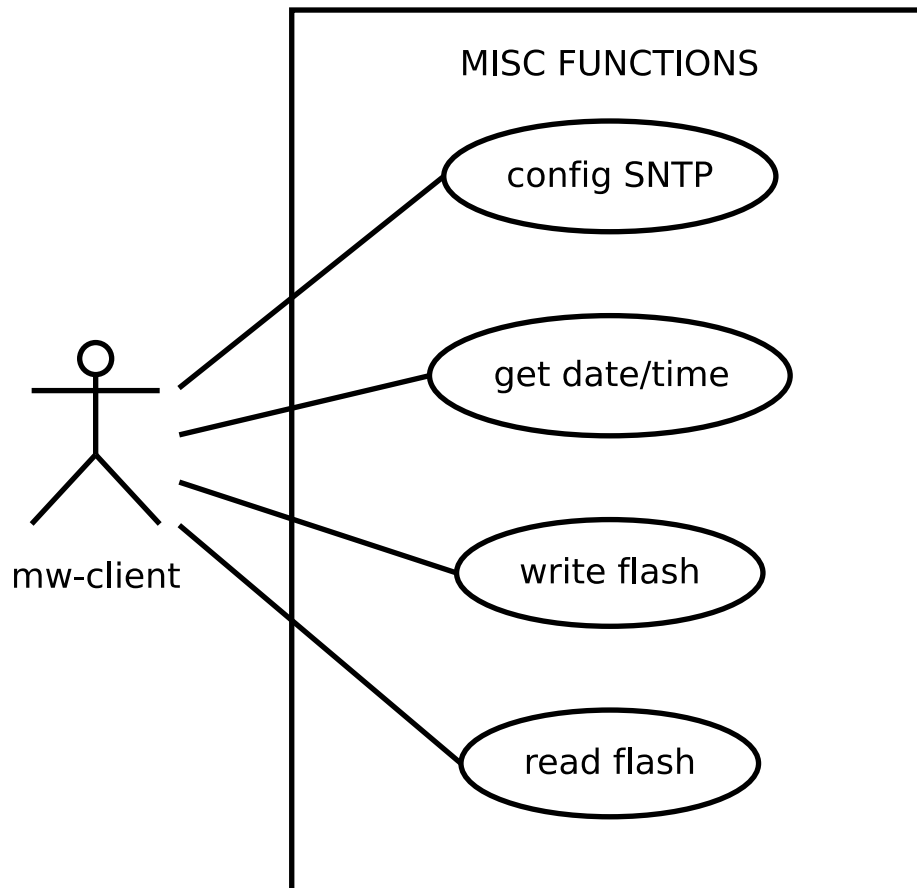


Figure 9: Misc functions

- 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 10.

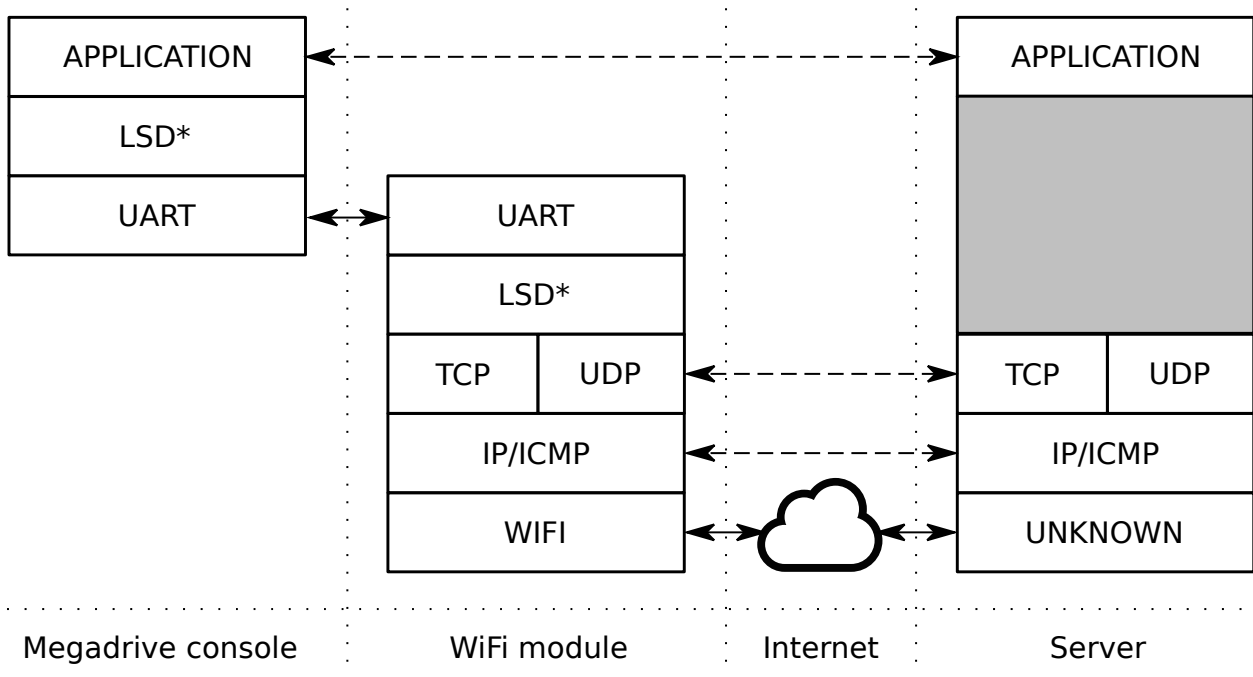


Figure 10: MegaWiFi Communications flow

- **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 10, 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. The in-cart UART uses a 24 MHz crystal oscillator, limiting the maximum baud rate to 1.5 Mbps. As the bandwidth is relatively 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.

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 10).

### 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.

LSD protocol data units (PDUs) have the format shown on Figure 11:

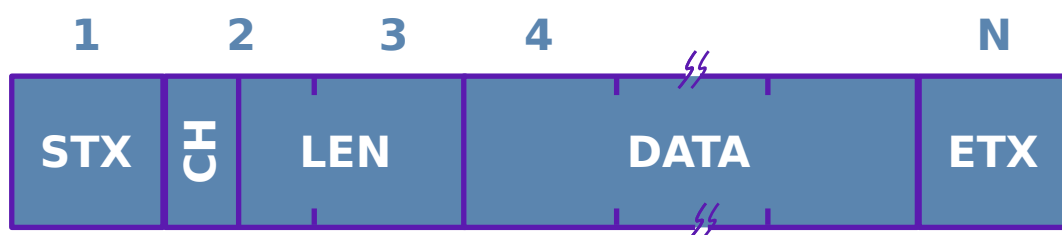


Figure 11: LSD PDU format

- 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 12 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 12 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.

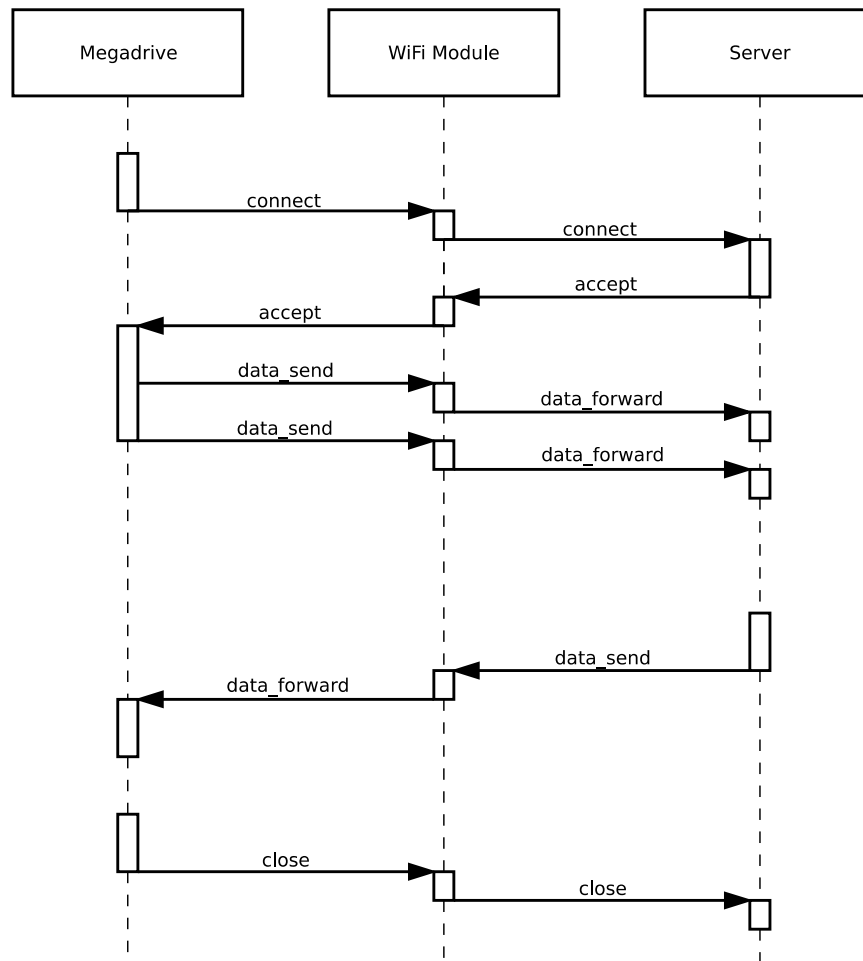


Figure 12: Typical TCP socket usage sequence

The first step when using TCP sockets, is to establish the connection with the server. Figure 12 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 12 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 12, 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

**NOTE:** version 1.0 of the firmware, does not yet support 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.

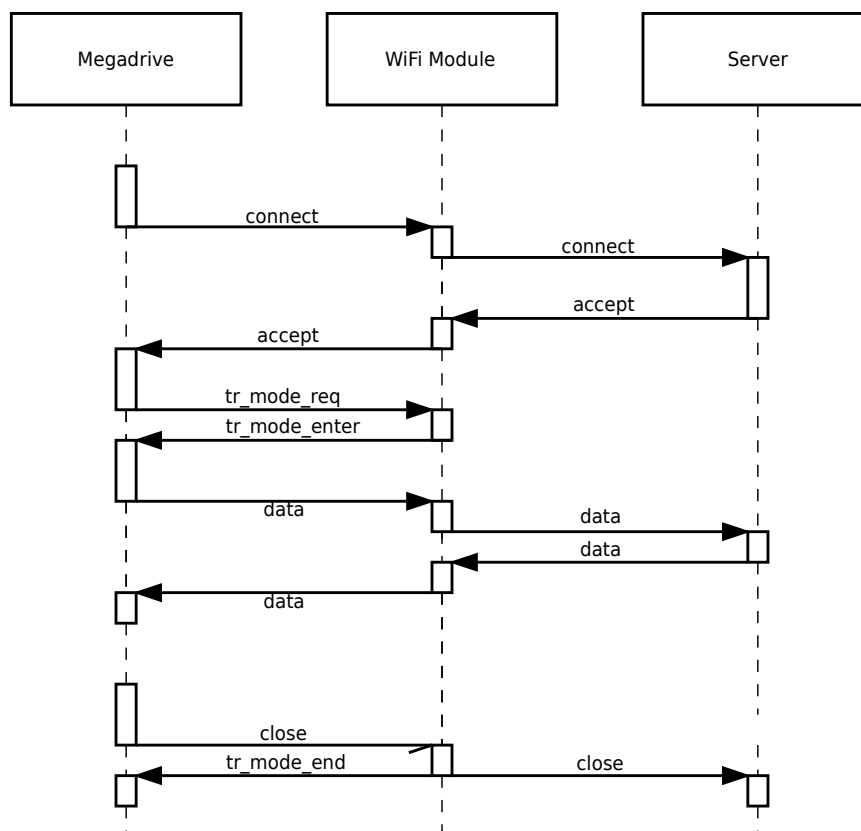


Figure 13: Transparent mode usage sequence

Figure 13 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/UDP/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 13, 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 14.

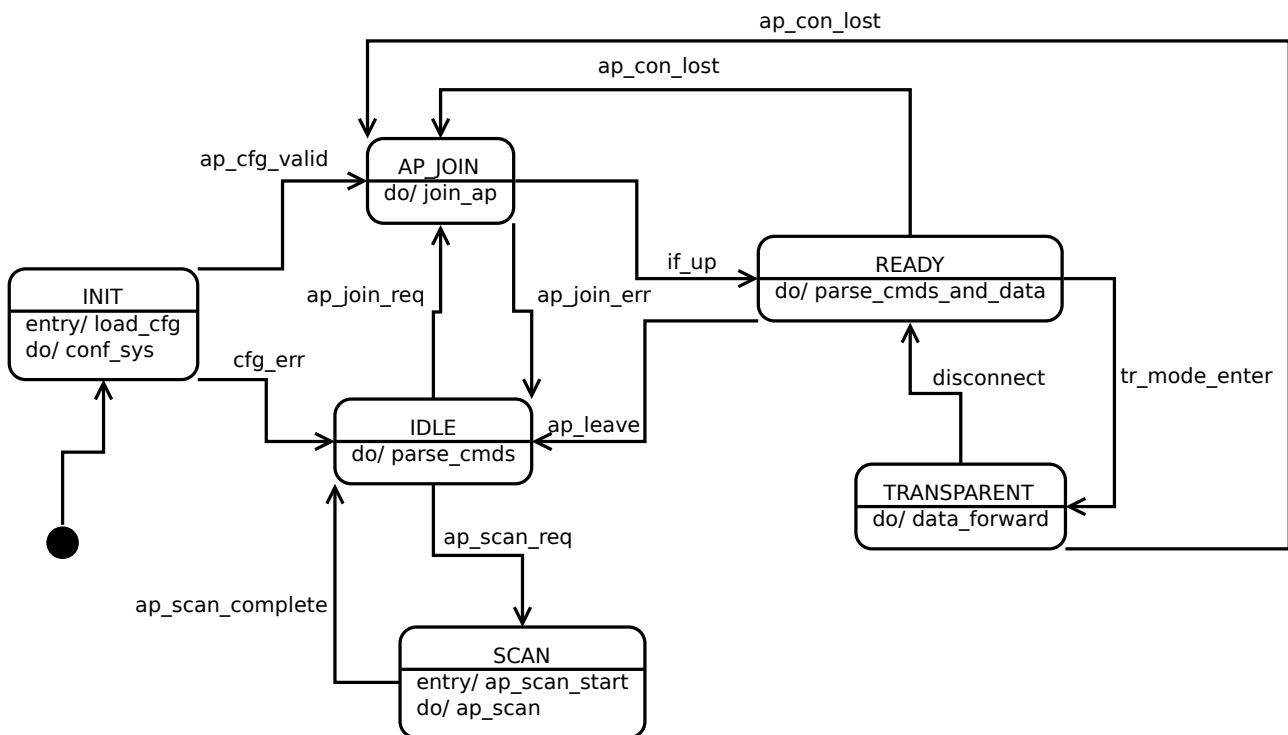


Figure 14: MegaWiFi finite state machine

Events shown on Figure 14 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 15. 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.

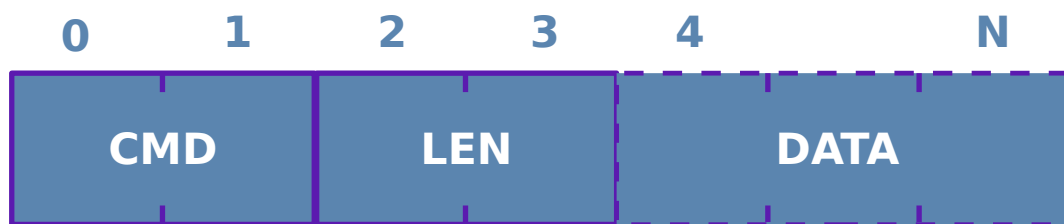


Figure 15: MegaWiFi API command format

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_CURRENT	6	Get the current IPv4 configuration
RESERVED		
IP_CFG	8	Configure IPv4 network
IP_CFG_GET	9	Get IPv4 network configuration
DEF_AP_CFG	10	Set default Access Point configuration slot
DEF_AP_CFG_GET	11	Get default Access Point configuration slot
AP_JOIN	12	Join an access point
AP_LEAVE	13	Leave a previously joined access point
TCP_CON	14	Create a TCP socket and try establishing a connection
TCP_BIND	15	Create a TCP socket and bind it to a port
RESERVED		
CLOSE	17	Disconnect a TCP connected socket
UDP_SET	18	Create a UDP socket and bind it to a port
RESERVED		
SOCK_STAT	20	Obtains TCP/UDP socket status
PING	21	Ping request
SNTP_CFG	22	Configure SNTP service
SNTP_CFG_GET	23	Get SNTP configuration
DATETIME	24	Get date and time
DT_SET	25	Set date and time
FLASH_WRITE	26	Write to WiFi module flash
FLASH_READ	27	Read from WiFi module flash
FLASH_ERASE	28	Erase sector from WiFi module flash
FLASH_ID	29	Get WiFi module flash identifier
SYS_STAT	30	Get system status
DEF_CFG_SET	31	Set default configuration
HRNG_GET	32	Get random numbers
BSSID_GET	33	Get the WiFi SSID from specified interface

Command	Code	Description
GAMERTAG_SET	34	Set player Gamertag
GAMERTAG_GET	35	Get player Gamertag
LOG	36	Print string to WiFi module log trace
FACTORY_RESET	37	Reset WiFi configuration to factory defaults
SLEEP	38	Set the module to low power sleep mode
HTTP_URL_SET	39	Set URL for HTTP requests
HTTP_METHOD_SET	40	Set method for HTTP requests
HTTP_CERT_QUERY	41	Query x.509 hash of installed PEM certificate
HTTP_CERT_SET	42	Set PEM certificate for HTTPS requests
HTTP_HDR_ADD	43	Add HTTP header for HTTP requests
HTTP_HDR_DEL	44	Delete HTTP header previously added
HTTP_OPEN	45	Open connection and start HTTP request
HTTP_FINISH	46	Finish HTTP request and get response payload length
HTTP_CLEANUP	47	Cleanup HTTP request data
SERVER_URL_GET	48	Get URL of the default server
SERVER_URL_SET	49	Set URL of the default server
ERROR	255	Error command reply

### 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.

**Note:** This command can only be used when the system is on IDLE state.

- 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: 97 bytes.
  - Format:
    - Configuration number (1 byte): The same as the one used on the request.
    - 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\_CURRENT*

Get current IPv4 configuration: IP address, network mask, gateway and DNS addresses.

- Payload length: 0
- Reply:
  - OK: IP configuration obtained.
    - Length: 20 bytes.
    - Format: IP address (4 bytes), netmask (4 bytes), gateway (4 bytes), DNS 1 (4 bytes), DNS 2 (4 bytes).

### 3.4.7 *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): 1<sup>st</sup> domain name server IPv4 address in binary format.
  - DNS2 (4 bytes): 2<sup>nd</sup> domain name server IPv4 address in binary format.
- Reply:

- OK: IPv4 configuration successfully set.
  - Length: 0 bytes.
- ERROR: Could not set specified IPv4 configuration.

### 3.4.8 *IP\_CFG\_GET*

Obtain specified IPv4 configuration. Read section 3.4.7 for more details.

- Payload length: 1 byte.
- Format:
  - Configuration number (1 byte): from 0 to 2.
- Reply:
  - OK: Requested IPv4 configuration successfully obtained.
    - Length: 24 bytes
    - Format:
      - Configuration number (1 byte): The same as the one used on the request.
      - 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): 1<sup>st</sup> domain name server IPv4 address in binary format.
      - DNS2 (4 bytes): 2<sup>nd</sup> domain name server IPv4 address in binary format.
  - ERROR: Could not get requested configuration.

### 3.4.9 *DEF\_AP\_CFG*

Sets the default AP configuration slot.

- Payload length: 1 byte.
- Format:
  - AP configuration slot number (1 byte): from 0 to 2.
- Reply:
  - OK: Successfully configured default AP slot.
    - Length: 0 bytes.

- ERROR: Could not configure default AP slot.

### 3.4.10 *DEF\_AP\_CFG\_GET*

Obtains the default AP configuration slot number.

- Payload length: 0
- Reply:
  - OK: Successfully obtained the default AP configuration slot.
    - Length: 1 byte
    - Format:
      - AP configuration slot number (1 byte): from 0 to 2.
  - ERROR: Could not obtain the default AP slot.

### 3.4.11 *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.

### 3.4.12 *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.

### 3.4.13 *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.

#### 3.4.14 *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.

#### 3.4.15 *CLOSE*

Closes a previously opened socket, and frees it. It also cancels and frees unconnected but bound sockets. Can be used for both TCP and UDP 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.

### 3.4.16 *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.

### 3.4.17 *SOCK\_STAT*

Gets status of the TCP/UDP socket associated with the requested channel. Also clears the event flag of the requested channel (if set, see section 3.4.27).

- Payload length: 1 byte.
- Format:
  - Channel: number of the channel to get status of.
- Reply:
  - OK: status obtained.

- Length: 1 byte.
- Format:
  - Socket status (1 byte): One of the following status codes:
    - MW SOCK\_NONE (0): Socket is unused or closed.
    - MW SOCK\_TCP\_LISTEN (1): TCP socket is listening for incoming connections.
    - MW SOCK\_TCP\_EST (2): A TCP connection is established on this socket.
    - MW SOCK\_UDP\_READY (3): UDP socket has been configured and is ready to send and receive datagrams.
  - ERROR: Could not get the socket status.

### 3.4.18 PING

As of version 1.0 of the firmware, this command is 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.

**Note:** this command requires a reboot of the WiFi module to take effect.

- Payload length: variable
- Format:
  - Null terminated TZ string (E.g. “UTC+1”).
  - Null terminated NTP server list (up to 3 servers).
  - Additional Null termination.
- Reply:
  - OK: SNTP configuration successfully applied.
    - Length: 0 bytes.
  - ERROR: Could not set requested SNTP configuration.

### 3.4.20 SNTP\_CFG\_GET

Obtains the configuration of the SNTP time synchronization client service.

- Payload length: 0 bytes
- Reply:

- OK: SNTP configuration successfully obtained.
  - Length: variable.
  - Format:
    - Null terminated TZ string
    - List of Null terminated NTP servers, up to 3.
    - Additional null termination.
- ERROR: Could not get SNTP configuration.

### 3.4.21 DATETIME

Obtains the date and time kept 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, in binary format.
      - Date Time (variable) : The date and time in text format. E.g.: “Thu Mar 3 12:26:51 2016”.

### 3.4.22 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.



### 3.4.23 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 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 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 0x2FFFFFFF.
  - Data: Data payload.
- Reply:
  - OK: data written to specified address.
    - Length: 0 bytes.
  - ERROR: could not write data to specified location.

### 3.4.24 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 0x00000000 and for the default 4 MiB chips, go up to 0x2FFFFFFF.
  - 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.

### 3.4.25 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 (2 bytes): 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.

### 3.4.26 FLASH\_ID

**NOTE:** This command is not supported from firmware version 0.5 and upper. Documentation is kept for reference.

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.

### 3.4.27 SYS\_STAT

Obtains system status flags.

- Payload length: 0 bytes.
- Reply:
  - OK: System status obtained:
    - Length: 4 bytes
    - Format: System status, as shown on figure 16. The meaning of each field is as follows:
      - STAT: System status, one of the following:
        - MW\_ST\_INIT (0): System is starting.
        - MW\_ST\_IDLE (1): System is not connected to the AP and waiting for commands.
        - MW\_ST\_AP\_JOIN (2): System is joining an Access Point.
        - MW\_ST\_SCAN (3): System is performing an AP scan.
        - MW\_ST\_READY (4): System is connected to the AP and ready to send/receive data.
        - MW\_ST\_TRANSPARENT (5): System is on transparent mode. This state is not likely going to be reported with this command, because while on transparent mode, command interpreter is disabled.
      - ONL: Online flag. If set, system is connected to the Internet.
      - CFG: Configuration flag. If set, configuration has successfully loaded from flash or set by a configuration command.
      - DTS: Date time set flag. If set, date and time has been synchronized with NTP servers at least once.
      - RESERVED: Currently unused. Might be used on future firmware versions.
      - CHEV: Channel event flags. Each set bit of this field indicates there is an unread event on its corresponding channel (i.e. bit 1 indicates there is an event on channel 1, bit 2 indicates there is an event on channel 2 and so on). To clear an event flag, SOCK\_STAT command (section 3.4.17) must be issued, with the channel number associated with the flag to clear.
  - ERROR: Could not get system status flags.

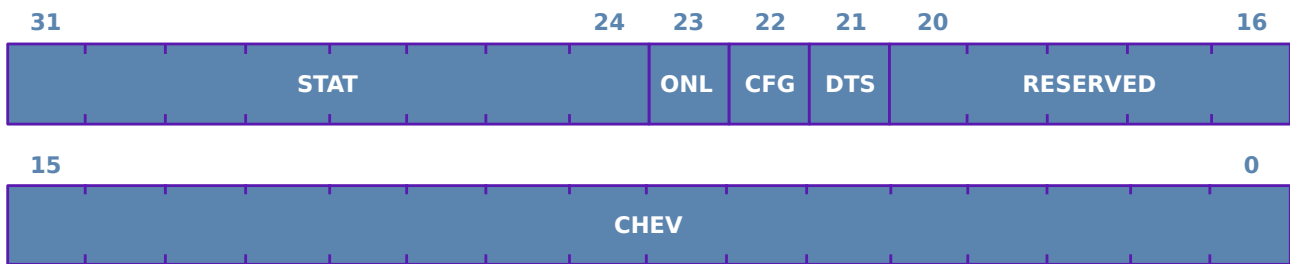


Figure 16: System status flags

### 3.4.28 DEF\_CFG\_SET

Sets configuration to factory default. **Note:** This command requires a WiFi module reset to take effect.

- Payload length: 4 bytes.
- Format
  - Magic number (4 bytes): 0xFEAA5501. Factory reset will fail if this value is wrong.
- Reply:
  - OK: Configuration reset to default. Reboot the module for the changes to take effect.
    - Length: 0 bytes.
  - ERROR: Could not set configuration to factory default.

### 3.4.29 HRNG\_GET

Gets an array of random numbers, generated using the hardware random number generator inside the WiFi module.

- Payload length: 2 bytes.
- Format
  - Random number length (2 bytes): Number of bytes of random numbers to generate.
- Reply:
  - OK: Requested amount of random numbers generated.
    - Length: variable (matching the random number length field on the request).
    - Format:
      - Random data (variable): Generated random number. The length of this field matches the specified on the random number length field of command request.
  - ERROR: Could not generate requested random data.

### 3.4.30 BSSID\_GET

Get the BSSID (MAC address) of the network interfaces of the WiFi module.

- Payload length: 1 byte.
- Format:
  - Interface number to get BSSID from: 0 for the STATION interface, 1 for the AP interface.
- Reply:
  - OK: Interface BSSID obtained successfully.
    - Length: 6 bytes.
    - Format:
      - The requested BSSID in binary format (6 bytes).
  - ERROR: Could not get requested BSSID.

### 3.4.31 GAMERTAG\_SET

Set gamertag information for requested slot.

- Payload length: 968 bytes.
- Format:
  - Gamertag configuration slot to set from 0 to 3 (1 byte).
  - Reserved (3 bytes).
  - Unique player id (4 bytes).
  - Nickname (32 bytes, zero padded).
  - Security (32 bytes, zero padded).
  - Tagline (32 bytes, zero padded).
  - Telegram token (64 bytes, zero padded).
  - 32x48 px avatar image tiles (768 bytes).
  - 16 color avatar palette (32 bytes).
- Reply:
  - OK: Gamertag configuration successfully set.
    - Length: 0 bytes.
  - ERROR: Could not set gamertag configuration.

### 3.4.32 *GAMERTAG\_GET*

Get gamertag information from specified slot.

- Payload length: 1 byte.
- Format:
  - Slot to get information from (1 byte).
- Reply:
  - OK: Got gamertag information.
    - Length: 964 bytes.
    - Format:
      - Unique player id (4 bytes).
      - Nickname (32 bytes, zero padded).
      - Security (32 bytes, zero padded).
      - Tagline (32 bytes, zero padded).
      - Telegram token (64 bytes, zero padded).
      - 32x48 px avatar image tiles (768 bytes).
      - 16 color avatar palette (32 bytes).
  - ERROR: Failed to get gamertag information.

### 3.4.33 *LOG*

Write string to MegaWiFi log trace.

- Payload length: variable.
- Format:
  - Null terminated string to write to log (variable length).
- Reply:
  - OK: Log written.
    - Length: 0 bytes.
  - ERROR: Could not write to log.

### 3.4.34 *FACTORY\_RESET*

Reset WiFi module configuration to factory defaults.

- Payload length: 0 bytes.

- Reply:
  - OK: Factory reset completed.
    - Length: 0 bytes.
- ERROR: Could not perform factory reset.

### 3.4.35 *SLEEP*

Put WiFi module to low-power sleep mode. The WiFi module is powered off, and requires a RESET cycle to be restarted. Use this command to save power if you will not use the module in a long time interval.

- Payload length: 0 bytes.
- Reply:
  - OK: Module is powered off.
    - Length: 0 bytes.
- ERROR: Could not perform sleep command.

### 3.4.36 *HTTP\_URL\_SET*

Set the URL for HTTP requests.

- Payload length: variable.
- Format:
  - Null terminated string corresponding to the desired URL (variable length).
- Reply:
  - OK: URL successfully set.
    - Length: 0 bytes.
  - ERROR: Could not set URL.

### 3.4.37 *HTTP\_METHOD\_SET*

Set method for HTTP requests.

- Payload length: 1 byte.
- Format:
  - Method index (1 byte), with the following code:
    - 0: GET
    - 1: POST

- 2: PUT
- 3: PATCH
- 4: DELETE
- 5: HEAD
- 6: NOTIFY
- 7: SUBSCRIBE
- 8: UNSUBSCRIBE
- 9: OPTIONS
- Reply:
  - OK: Method set successfully.
    - Length: 0 bytes.
  - ERROR: Could not set HTTP method.

### 3.4.38 *HTTP\_CERT\_QUERY*

Get the x.509 hash of the installed PEM certificate.

- Payload length: 0 bytes.
- Reply:
  - OK: Certificate hash obtained.
    - Length: 4 bytes.
    - Format:
      - Certificate hash in binary form. 0xFFFFFFFF if no certificate is installed or error occurs.
  - ERROR: Could not get certificate hash data.

### 3.4.39 *HTTP\_CERT\_SET*

Prepare the module to set the x.509 PEM certificate to use in HTTPS requests. On success, the certificate must be sent as regular data, using the reserved HTTP\_CH channel.

- Payload length: 6 bytes.
- Format:
  - x.509 certificate hash (4 bytes).
  - Certificate length (2 bytes).
- Reply:



- OK: Module is ready to receive the certificate. It must be immediately sent using the HTTP\_CH reserved channel.
  - Length: 0 bytes.
- ERROR: Failed to prepare the module to set the certificate.

#### 3.4.40 HTTP\_HDR\_ADD

Add an HTTP header to the request.

- Payload length: variable
- Format:
  - Null terminated header key (e.g. “Content-Type”).
  - Null terminated header value (e.g. “application/json”).
- Reply:
  - OK: Header successfully added.
    - Length: 0 bytes.
  - ERROR: Could not add HTTP header.

#### 3.4.41 HTTP\_HDR\_DEL

Delete a previously added HTTP header.

- Payload length: variable.
- Format:
  - Null terminated header key (e.g. “Authorization”).
- Reply:
  - OK: Header successfully removed.
    - Length: 0 bytes.
  - ERROR: Could not delete specified header.

#### 3.4.42 HTTP\_OPEN

Start the previously configured HTTP request. This command causes the client to send the HTTP request to the server. If a data payload is required for the request, the payload length is configured in this request. After completing the command, if a data payload has been specified, it must be sent as regular data, using the HTTP\_CH reserved channel.

- Payload length: 4 bytes.
- Format:

- Length of the payload (if any) to send after completing this command (4 bytes).
- Reply:
  - OK: HTTP request sent to the server. If a payload has been specified, module is ready to receive the payload data.
    - Length: 0 bytes.
  - ERROR: Error occurred while starting the request.

#### 3.4.43 *HTTP\_FINISH*

Finishes an HTTP request, previously started with the HTTP\_OPEN command.

- Payload length: 0 bytes.
- Reply:
  - OK: HTTP request finished. If a reply data payload is specified, it must be received as normal data using HTTP\_CH reserved channel, just after completing this command.
    - Length: 6 bytes.
    - Format:
      - Payload data length to receive after completing the command (4 bytes).
      - HTTP request status code (e.g. 200, 404, 500, etc.) (2 bytes).
  - ERROR: Could properly finish the HTTP request.

#### 3.4.44 *HTTP\_CLEANUP*

Clean up an HTTP request, freeing associated resources.

- Payload length: 0 bytes.
- Reply:
  - OK: HTTP resources freed.
    - Length: 0 bytes.
  - ERROR: Failed to free HTTP request associated resources.

#### 3.4.45 *SERVER\_URL\_GET*

Get the server URL used by default for several services.

- Payload length: 0 bytes.
- Reply:
  - OK: Server URL obtained.

- Length: variable.
- Format:
  - Null terminated default server URL.
- ERROR: Could not get default server URL.

### 3.4.46 *SERVER\_URL\_SET*

Sets the default server URL used for several services.

- Payload length: variable.
- Format:
  - Null terminated default server URL.
- Reply:
  - OK: Default server URL properly set.
    - Length: 0 bytes.
  - ERROR: Failed to set server URL.

## 4 MegaWiFi API for SEGA MegaDrive/Genesis

### 4.1 Introduction

MegaWiFi API can be used to greatly ease coding MegaWiFi enabled programs on the Megadrive. It avoids having to handle binary protocols and commands as explained in section 3.

### 4.2 API documentation

The full API documentation [can be found here](#), along with an example program using it. There you can find the most up to date version of this chapter, along with the conveniently Doxygen generated documentation of every function, data type and module. Unless you cannot reach a computer right now, it might be a good idea stopping reading here, and opening the documentation linked above in your browser of choice.

### 4.3 Building

You will need a complete Genesis/Megadrive toolchain. The sources use some C standard library calls, such as `memcpy()`, `setjmp()`, `longjmp()`, etc. Thus your toolchain must include a C standard library implementation such as *newlib*.

To build the files, you can use the provided Makefile, suiting it to your needs, or just add the source files to your project to build them.

### 4.4 Overview

The MegaWiFi API consists of the following modules:

- `loop`: Loop handling for single threaded Megadrive programs.
- `mpool`: Simple memory pool implementation.
- `megawifi`: Communications with the WiFi module and the Internet, including sockets and HTTP/HTTPS.
- `mw-msg`: MegaWiFi command message definitions.
- `util`: General purpose utility functions and macros.

The `mw-msg` module contains the message definitions for the different MegaWiFi commands and command replies. Fear not because usually you do not need to use this module, unless you are doing something pretty advanced not covered by the `megawifi` module API.

The `util` module contains general purpose functions and macros not fitting in the other modules, such as `ip_validate()` to check if a string is a valid IP address, `str_to_uint8()` to convert a string to an 8-bit number, etc.

The other modules (`loop`, `mpool`, `megawifi`) are covered in depth below.

There is also a `json` module which includes `jsmn` library along with some helper functions to parse JSON formatted strings. Please read `jsmn` documentation to learn how its tokenizer works.

#### 4.4.1 Loop module

This module implements the main loop of the program. It allows easily adding and removing functions to be run on the main loop, as well as timers based on the frame counter. It also provides a hacky interface to perform pseudo synchronous calls (through the `loop_pend()` and `loop_post()` semantics) without disturbing the loop execution.

A typical Megadrive game contains a main loop with a structure similar to this:

```
void main(void)
{
    // Perform initialization
    init();

    // Infinite loop with game logic
    while(1) {
        wait_vblank();
        draw_screen();
        play_sound();
        read_input();
        game_logic();
    }
}
```

The game performs the initialization using `init()` function, and then enters an infinite loop that:

1. Waits for the vertical blanking period to begin.
2. Updates the frame (scroll, sprites, tiles, etc).
3. Keeps the music and SFX playing.
4. Reads controller inputs.
5. Computes game logic, such as collision detection, player/enemy movements, etc. When this step finishes, we have all the data to draw the next frame.

The order of these elements might be slightly different, but these are the usual suspects in loop game design.

On the contrary, the recommended way to write a MegaWiFi program requires using the `loop` module to implement the main loop. When using MegaWiFi API, the code above should be written like this (my recommendation with these examples is that you read the code from the bottom function to the top ones):

```
#include "mw/util.h"
#include "mw/loop.h"

#define MW_MAX_LOOP_FUNCS 2
```

```

#define MW_MAX_LOOP_TIMERS 4

// Run once per frame
static void frame_cb(struct loop_timer *t)
{
    // Avoid compiler warning because unused t parameter
    UNUSED_PARAM(t);

    draw_screen();
    play_sound();
    read_input();
    game_logic();
}

static void main_loop_init(void)
{
    static struct loop_timer frame_timer = {
        .timer_cb = frame_cb,
        .frames = 1,
        .auto_reload = TRUE
    };

    loop_init(MW_MAX_LOOP_FUNCS, MW_MAX_LOOP_TIMERS);
    loop_timer_add(&frame_timer);
}

static void init(void)
{
    // Initialize game stuff
    // ...

    // Initialize game loop
    main_loop_init();
}

void main(void)
{
    // Initialization
    init();

    loop();
    // Function above should never return
}

```

The `init()` function now calls `main_loop_init()` to:

1. Initialize the loop module by calling `loop_init()`.
2. Add a `loop_timer` that runs `frame_cb()` callback once per frame, ideally at the beginning of the vertical blanking period.

As `frame_cb()` is run once per frame, we can (and we **must**) remove the `wait_vblank()` function call, and add all the remaining code previously inside the `while(1)` to the `frame_cb()`.

Using this module, now if you for example want to add another `loop_timer` running each 5 frames to update the background animation, you just have to create the callback and the `loop_timer` structure, to finally add it to the loop by calling `loop_timer_add()`.

In addition to timers, the module also allows adding functions that are run the spare frame time. We will see this in greater detail when talking about the `megawifi` module.

The `loop` module also implements another functionality: pseudo-synchronous event waiting. This eases avoiding the typical callback hell that occurs when coding asynchronous programs. The pseudo-synchronous waits work like this:

1. The function that wants to wait for an event, calls `loop_pend()` function. The execution of the function is then suspended.
2. The suspended function is resumed by doing a `loop_post()` call from other point of the code.

The important thing to take into account, is that while a function is suspended on a `loop_pend()` call, the other loop functions and loop timers continue running undisturbed. Isn't this neat?

Nevertheless, you have to be careful when using `loop_pend()/loop_post()`: If you nest several `loop_pend()` calls, the following `loop_post()` calls will resume suspended functions in the reverse order of the `loop_pend()` calls. This is not probably what you want, and thus nesting `loop_pend()` calls is discouraged unless you know what you are doing.

Also when using the `loop` module, you have to be careful not to block a loop function or loop timer. These functions must do their task quickly and exit as fast as possible. Otherwise, if you block by polling (e.g waiting for `vblank`, or waiting for the player to press a button), other loop functions or loop timers will not be able to get any CPU time. The only allowed way to block a loop function or loop timer, is by calling `loop_pend()` function (or any other function that uses it, such as `mw_sleep()`).

As using the `loop` module seems to make the code more complex, maybe you are wondering why bothering with it. We will come to that later.

#### 4.4.2 *Mpool module*

This module implements a very fast and simple memory pool for dynamic memory allocation. Allocated memory is obtained from the unused region between the end of the `.bss` section and the stack top. The implementation is pretty simple: an internal pointer grows when memory is requested using `mp_alloc()`, and is reset to the specified position to free memory using `mp_free_to()`. This restricts the usage of the module to scenarios that free memory in exactly the reverse order in which they requested it (it does not allow generic allocate/free such as `malloc()` does).

One thing interesting about this module is that you can free the memory allocated by several `mp_alloc()` calls with a single `mp_free_to()` call. This behavior can be nasty if you are accustomed to the usual *one free() per malloc()* scheme, but it is sometimes handy. For example, it helps avoiding memory fragmentation and memory leaks when changing from one game level to another. Imagine that you have two game levels and they allocate memory for several structures:

```

void level1_init(void)
{
    struct level1_data *l1d = mp_alloc(sizeof(struct level1_data));
    struct enemy *enem = mp_alloc(L1_NUM_ENEMIES * sizeof(struct enemy));
}

void level2_init(void)
{
    struct level2_data *l2d = mp_alloc(sizeof(struct level2_data));
    struct enemy *enem = mp_alloc(L2_NUM_ENEMIES * sizeof(struct enemy));
}

```

Imagine also that the game allocates more memory during level play for other purposes (bullets, explosions, etc). Now you finish the level and want to make sure all the memory is freed to load the new level. With `malloc()/free()` you would need to track each allocation and do the corresponding `free()`. But with the `mpool` module it is way easier:

```

void level1_deinit(void)
{
    mp_free_to(l1d);
}
void level2_deinit(void)
{
    mp_free_to(l2d);
}

```

And that's all, the call to `mp_free_to(l1d)` will deallocate all the memory obtained since the call to the first `mp_alloc()` in `level1_init()`, and you make sure no memory fragmentation and no memory leaks are caused by all the allocations in the level.

### 4.4.3 Megawifi module

And we finally arrive to the `megawifi` module API. This API allows of course sending and receiving data to/from the Internet, along with some more functions such as:

- Scanning APs, associating and disassociating to/from them.
- Creating both client and server network sockets.
- Performing HTTP/HTTPS client requests.
- Reading and writing from/to the non-volatile flash memory in the WiFi module.
- Keeping the time and day, accurately synchronized to NTP servers.
- Generating large amounts of random numbers blazingly fast.

You can use this API the hard way (directly sending commands defined in `mw-msg`), or the easy way (through the API calls in `megawifi`). Of course the latter is recommended.

Most API functions require sending and receiving data to/from the WiFi module. But the data send/reception is decoupled from the command functions: the API functions prepare the module to send/receive data, but the data is not sent/received until the `mw_process()` function is called. As



the `mw_process()` function polls the WiFi module for data, it is advisable to run it as frequently as possible. The easiest way to do this, is using a *loop\_func*. Just set up a *loop\_func* running `mw_process()` to ensure this function will be continuously executed, and you're done:

```
#include "mw/util.h"
#include "mw/loop.h"
#include "mw/megawifi.h"

static void megawifi_loop_cb(struct loop_func *f)
{
    UNUSED_PARAM(f);
    mw_process();
}

static void main_loop_init(void)
{
    // Loop initialization code
    // [...]
    static struct loop_func megawifi_loop = {
        .func_cb = megawifi_loop_cb
    };
    loop_func_add(&megawifi_loop);
}
```

Using a *loop\_func* like this makes sending and receiving data during game idle time way easier. Just make sure you set up your loops properly, and leave a bit of time for the loop function running `mw_process()`. Otherwise this function will starve and no data will be sent/received!

About the API calls, basically all of them are synchronous or pseudo-synchronous, excepting the following ones, that are asynchronous and use callbacks to signal task completion:

- `mw_send()`: Send data to the other socket end.
- `mw_recv()`: Receive data from the other socket end.
- `mw_cmd_send()`: Send a command to the WiFi module.
- `mw_cmd_recv()`: Receive a command reply from the WiFi module.

Usually `mw_cmd_send()` and `mw_cmd_recv()` are not needed unless you decide to go down the hard path (using `mw-msg` to build commands yourself). For sending/receiving data, it's up to you using the asynchronous `mw_send()` and `mw_recv()` or their pseudo-synchronous counterparts `mw_send_sync()` and `mw_recv_sync()`.

To save precious RAM, command functions reuse the same buffer. Thus when a command reply is obtained, you have to copy the needed data from the buffer before issuing another command. Otherwise the data in the previously received buffer will be lost.

## 4.5 Putting all together

In this section several examples explaining how to code typical tasks are presented.

### 4.5.1 Connection configuration

MegaWiFi modules have 3 configuration slots, allowing to store 3 different network configurations. The configuration parameters are:

- Access point configuration (SSID, password), using `mw_ap_cfg_set()`. This usually requires a previous AP scan using `mw_ap_scan()` and cycling through scan results using `mw_ap_fill_next()`.
- IP configuration, using `mw_ip_cfg_set()`. Both automatic (DHCP) and manual configurations are supported.

The good news is that you do not need to code the connection configuration, you can use the [wflash bootloader](#) to configure the network. As the configuration is stored inside the module, you can use it from your game, even if you delete the wflash bootloader ROM from the MegaWiFi cartridge.

### 4.5.2 Program initialization

Basically you have to initialize megawifi and the game loop as explained before. You also have to create a `loop_func` to run `mw_process()` and a `loop_timer` with a 1 frame period to handle the game loop. The code below shows how to do this, and also how to detect if the WiFi module is installed, along with its firmware version.

```
#include "mw/util.h"
#include "mw/mpool.h"
#include "mw/loop.h"
#include "mw/megawifi.h"

// Length of the wflash buffer
#define MW_BUFLen 1440

// TCP port to use (set to Megadrive release year ;-))
#define MW_CH_PORT 1985

// Maximum number of loop functions
#define MW_MAX_LOOP_FUNCS 2

// Maximum number of loop timers
#define MW_MAX_LOOP_TIMERS 4

// Command buffer
static char cmd_buf[MW_BUFLen];

// Runs mw_process() during idle time
static void idle_cb(struct loop_func *f)
{
    UNUSED_PARAM(f);
    mw_process();
}

// MegaWiFi initialization
static void megawifi_init_cb(struct loop_func *f)
```

```

{
    uint8_t ver_major = 0, ver_minor = 0;
    char *variant = NULL;
    enum mw_err err;

    // megawifi_init_cb is run only once. Use idle_cb from now on
    f->func_cb = idle_cb;

    // Initialize MegaWiFi
    mw_init(cmd_buf, MW_BUFLen);

    // Try detecting the module
    err = mw_detect(&ver_major, &ver_minor, &variant);

    if (MW_ERR_NONE != err) {
        // Megawifi cart not found!
        // [...]
    } else {
        // MegaWiFi found!
        // [...]
    }
}

// Run the game loop once per frame
static void frame_cb(struct loop_timer *t)
{
    UNUSED_PARAM(t);

    // One iteration of game loop
    draw_screen();
    play_sound();
    read_input();
    game_logic();
}

// Loop initialization
static void main_loop_init(void)
{
    static struct loop_timer frame_timer = {
        .timer_cb = frame_cb,
        .frames = 1,
        .auto_reload = TRUE
    };
    static struct loop_func megawifi_loop = {
        .func_cb = megawifi_init_cb
    };

    loop_init(MW_MAX_LOOP_FUNCS, MW_MAX_LOOP_TIMERS);
    loop_timer_add(&frame_timer);
    loop_func_add(&megawifi_loop);
}

// Global initialization
static void init(void)
{
    // Initialize hardware and game
    // [...]
    // Initialize memory pool

```

```

    mp_init(0);
    // Initialize game loop
    main_loop_init();
}

/// Entry point
void main(void)
{
    // Initialization
    init();

    loop();
    // loop() should never return
}

```

### 4.5.3 Associating to an AP

Once configured, associating to an AP is easy. Just call `mw_ap_assoc()` with the desired configuration slot, and the module will start the process. You can wait until the association is successful or fails (because of timeout) by calling `mw_ap_assoc_wait()`. The following code tries to associate to an AP during 30 seconds (*fps* must be set previously to 60 on NTSC machines or 50 on PAL machines).

```

enum mw_err err;

err = mw_ap_assoc(slot);
if (MW_ERR_NONE == err) {
    err = mw_ap_assoc_wait(30 * fps);
}
if (MW_ERR_NONE == err) {
    // Association succeeded
} else {
    // Association failed
}

```

Once association has succeeded, you can try connecting to a server, or creating a server socket. DNS service will also start automatically after associating to the AP, but it takes a little bit more time. So if you need to use DNS just after associating to an AP, you should wait an additional second, e.g. by calling `mw_sleep(MS_TO_FRAMES(1000))`.

### 4.5.4 Connecting to a TCP server

Connecting to a server is straightforward: just call `mw_tcp_connect()` with the channel to use, the destination address (both IPv4 addresses and domain names are supported), the destination port, and optionally the origin port (if NULL, it will be automatically set):

```

enum mw_err err;

err = mw_tcp_connect(1, "www.duck.com", "443", NULL);
if (MW_ERR_NONE == err) {
    // Connection succeeded
} else {

```

```
    // Connection failed
}
```

Once connected, you can start sending and receiving data. When no longer needed, remember to close the connection with `mw_tcp_disconnect()`. The channel number must be from 1 to `LSD_MAX_CH - 1` (usually 2). The used channel number will be passed to all the calls relative to the connected socket (think about it like a socket number).

#### 4.5.5 Creating a TCP server socket

Creating a TCP server socket requires binding it to a port, using `mw_tcp_bind()`. After this, MegaWiFi will automatically accept any incoming connection on this port. You can check when the connection has been established by calling `mw_sock_conn_wait()`:

```
enum mw_err err;

err = mw_tcp_bind(1, 1985);
if (MW_ERR_NONE == err) {
    // Wait up to an hour for an incoming connection
    err = mw_sock_conn_wait(1, 60 * 60 * fps);
}
if (MW_ERR_NONE == err) {
    // Incoming connection established
} else {
    // Timeout, no connection established
}
```

#### 4.5.6 Sending data

You can send data once a connection has been established. The easiest way is using the synchronous variant, but as it suspends the execution of the calling function until data is sent, sometimes the asynchronous version is more convenient. The following code shows how to send *data* buffer of *data\_length* length using channel 1, with a two second timeout:

```
enum mw_err err;

err = mw_send_sync(1, data, data_length, 2 * fps);
if (MW_ERR_NONE == err) {
    // Data sent
} else {
    // Timeout, data was not sent
}
```

The same data can be sent this way using the asynchronous API:

```
void send_complete_cb(enum lsd_status stat, void *ctx)
{
    UNUSED_PARAM(ctx);

    if (LSD_STAT_COMPLETE == stat) {
        // Data successfully sent
    } else {
```

```

        // Sending data failed
    }
}

void send_example(void)
{
    enum lsd_status stat;

    stat = mw_send(1, data, data_length, NULL, send_complete_cb);
    if (stat < 0) {
        // Sending failed
    }
}

```

When using the asynchronous API, sometimes you do not need confirmation about when data has been sent. In that case, you do not need to use a completion callback, and can call `mw_send()` with this parameter set to `NULL`.

#### 4.5.7 Receiving data

You can receive data once a connection has been established. The easiest way is using the synchronous variant, but as it suspends the execution of the calling function until data is received, sometimes the asynchronous version is more convenient. The following code shows how to receive data buffer of `buf_length` maximum length using channel 1, with a 30 second timeout:

```

enum mw_err err;

err = mw_rcv_sync(1, data, &buf_length, 30 * fps);
if (MW_ERR_NONE == err) {
    // Data received
} else {
    // Failed to receive data
}

```

Note that when the function successfully returns, `buf_length` contains the number of bytes received.

The same data can be received this way using the asynchronous API:

```

void rcv_complete_cb(enum lsd_status stat, uint8_t ch, char *data,
                    uint16_t len, void *ctx)
{
    UNUSED_PARAM(ctx);

    if (LSD_STAT_COMPLETE == stat) {
        // Data successfully received
    } else {
        // Data reception failed
    }
}

void rcv_example(void)
{
    enum lsd_status stat;

```

```

stat = mw_recv(data, data_length, NULL, recv_complete_cb);
if (stat < 0) {
    // Reception failed
} else {
    // Data will be received by mw_process()
}
}

```

#### 4.5.8 Performing an HTTP/HTTPS request

megawifi module allows performing HTTP and HTTPS requests in a simple way. HTTP and HTTPS use the same API, the only difference is that setting an SSL certificate is required only if you want to use HTTPS. You can skip this step when using plain HTTP. Performing an HTTPS request requires the following steps. Some of them are optional and depend on the use case.

1. **(Optional)** Set the SSL certificate. This is only required when using HTTPS. You can retrieve the x509 hash of the currently stored certificate by calling `mw_http_cert_query()`. To set a different certificate, call `mw_http_cert_set()`. Once set, the certificate is stored on the non volatile memory, and will remain until replaced with a new one. Only one certificate can be stored at a time. Note you should not use this function unless required, because as it writes to Flash memory, it can wear the storage if used too often.
2. Set the URL (e.g. `https://www.example.com`). Use `mw_http_url_set()` for this purpose.
3. Set the HTTP method. Most commonly used ones are `MW_HTTP_METHOD_GET` and `MW_HTTP_METHOD_POST`. Use `mw_http_method_set()` to set it.
4. **(Optional)** add HTTP headers to the request. Many aspects of the requests can be controlled via headers. E.g. you can define the formatting of the data you are posting by adding the header “Content-type” with the mime type “text/html”, “application/json”, etc.
5. Open the connection. In this step, the HTTP or HTTPS connection is opened, and the request (including any headers added) is sent to the server. If the request contains a data payload, in this step the payload length is specified. The function `mw_http_open()` does this.
6. **(Optional)** send the request data payload (if any). This must be performed only if a payload length (greater than 0) was specified in the previous step. This is done the same way as sending data through sockets, with the `mw_send()` or `mw_send_sync()` functions, using the HTTP reserved channel (`MW_CH_HTTP`).
7. Finish the transaction. Call `mw_http_finish()` for the HTTP client to obtain the response to the request, along with its associated headers. If a response includes a data payload, its length is obtained in this step.
8. **(Optional)** if the previous step returned a reply payload length greater than 0, it must be received in this step by calling `mw_recv()` or `mw_recv_sync()` using the HTTP reserved channel (`MW_CH_HTTP`).

By looking to this list of steps, it might look complicated to perform an HTTP request, but the steps are relatively simple, and can be easily added to some functions. E.g., this code allows performing arbitrary GET (without payload) and POST (with JSON payload) requests:

```
// Performs initial steps of an HTTP request
static int http_begin(enum mw_http_method type, const char *url,
    unsigned int len) {
    enum mw_err err;

    err = mw_http_url_set(url);
    if (!err) {
        err = mw_http_method_set(type);
    }
    if (!err) {
        err = mw_http_open(len);
    }

    return err;
}

// Tries to synchronously receive exactly the indicated data length
static int sync_recv(uint8_t ch, char *buf, int len, uint16_t tout_frames)
{
    int recvd = 0;
    int err = 0;
    uint8_t get_ch = ch;
    int16_t get_len;

    while (err == 0 && recvd < len) {
        get_len = len - recvd;
        err = mw_recv_sync(&get_ch, buf + recvd, &get_len, tout_frames);
        if (!err) {
            if (get_ch != ch) {
                err = -1;
            } else {
                recvd += get_len;
            }
        }
    }

    return err;
}

// Performs final steps of an HTTP request
static int http_finish(char *recv_buf, unsigned int *len)
{
    enum mw_err err;
    uint32_t content_len = 0;

    err = mw_http_finish(&content_len, MS_TO_FRAMES(60000));
    err = err >= 200 && err <= 300 ? 0 : err;
    if (content_len > b.msg_buf_len) {
        err = -1;
    }
    if (!err && content_len) {
        err = sync_recv(MW_HTTP_CH, recv_buf, content_len, 0);
    }
    if (!err && len) {

```



```

        *len = content_len;
    }

    return err;
}

/*****
 * \brief Generic HTTP GET request without data payload.
 *
 * \param[in] url      URL for the request.
 * \param[out] recv_buf Buffer used for HTTP response data.
 * \param[out] len      Length of the response.
 *
 * \return HTTP status code, or -1 if request was not completed.
 *****/
int http_get(const char *url, char *recv_buf, unsigned int *len)
{
    enum mw_err err;

    err = http_begin(MW_HTTP_METHOD_GET, url, 0);
    if (!err) {
        err = http_finish(recv_buf, len);
    }

    return err;
}

/*****
 * \brief Generic POST request with JSON data payload.
 *
 * \param[in] url      URL for the request.
 * \param[in] data      Data to send in the POST data payload.
 * \param[in] length    Length of the data to send.
 * \param[in] content_type Content-Type HTTP header for the data payload.
 * \param[out] recv_buf Buffer used to receive reply data.
 * \param[inout] recv_len On input, length of recv_buf. On output, length
 *                        of the received reply data.
 *
 * \return HTTP status code or -1 if the request was not completed.
 *****/
int http_post(const char *url, const char *data, int length,
              const char *content_type, char *recv_buf,
              unsigned int *recv_len)
{
    enum mw_err err;

    // If this function fails, it is not critical
    mw_http_header_add("Content-Type", content_type);

    err = http_begin(MW_HTTP_METHOD_POST, url, length);
    if (!err) {
        err = mw_send_sync(MW_HTTP_CH, data, length, 0);
    }
    if (!err) {
        err = http_finish(recv_buf, recv_len);
    }

    return err;
}

```

```
}
```

In case you want HTTPS, you can set a PEM formatted certificate as follows:

```
void http_cert_set(const char *cert, int cert_len, uint32_t cert_hash)
{
    uint32_t hash = mw_http_cert_query();
    if (hash != cert_hash) {
        mw_http_cert_set(cert_hash, cert, cert_len);
    }
}
```

This function only sets the certificate if it has not been previously stored, avoiding to unnecessarily wear the Flash memory. To obtain a correct certificate hash, you can use openssl:

```
$ openssl x509 -hash in <cert_file_name> -noout
```

#### 4.5.9 Getting the date and time

MegaWiFi allows to synchronize the date and time to NTP servers. It is important to note that on console power up, the module date and time will be incorrect and should not be used. For the date and time to be synchronized, the module must be associated to an AP with Internet connectivity. Once associated, the date and time is automatically synchronized. The synchronization procedure usually takes only a few seconds, and once completed, date/time should be usable until the console is powered off.

To know if the date and time is in sync, you can use the `mw_sys_stat_get()` command:

```
union mw_msg_sys_stat *sys_stat;

sys_stat = mw_sys_stat_get();
if (sys_stat && sys_stat->dt_ok) {
    // Date and time synchronized
} else {
    // Date and time **not** synchronized, or other error
}
```

Once date and time is synchronized, you can get it, both in human readable format, and in the number of seconds elapsed since the epoch, with a single call to `mw_date_time_get()`:

```
char *date_time_string;
uint32_t date_time_bin[2];

date_time_string = mw_date_time_get(date_time_bin);
```

#### 4.5.10 Setting and getting gamertag information

MegaWiFi API allows to store and retrieve up to 3 gamertags. The gamertag information is contained in the `mw_gamertag` structure. This structure holds the gamertag unique identifier, nickname, security credentials (password) and a 32x48 avatar (tile information and palette). This

example shows how to set a gamertag (excepting the graphics data):

```
void gamertag_set(int slot, int id, const char *name,
                  const char *security, const char *tagline)
{
    struct mw_gamertag gamertag = {};
    struct mw_err err;

    gamertag.id = id;
    strcpy(gamertag.nickname, name);
    strcpy(gamertag.security, security);
    strcpy(gamertag.tagline, tagline);

    err = mw_gamertag_set(slot, &gamertag);
    if (MW_ERR_NONE == err) {
        // Gamertag set successfully
    } else {
        // Setting gamertag failed
    }
}
```

To read the gamertag, just call `mw_gamertag_get()` function, and the information corresponding to the requested slot will be returned:

```
struct mw_gamertag *gamertag = mw_gamertag_get(slot);

if (!gamertag) {
    // Something went wrong
} else {
    // Success!
}
```

#### 4.5.11 Reading the module BSSIDs

The module has two network interfaces, each one with its unique BSSID (MAC address). One interface is used for the station mode, while the other is for the AP mode. Each BSSID is 6-byte long. Currently the API does not allow using the AP mode, so to get the station mode BSSID you can do as follows:

```
uint8_t *bssid = mw_bssid_get(MW_IF_STATION);

if (!bssid) {
    // Something went wrong
} else {
    // Success! You can use bssid[0] through bssid[5]
}
```

#### 4.5.12 Reading and writing to non-volatile Flash

In addition to the standard 32 megabits of Flash ROM memory connected to the Megadrive 68k bus, MegaWiFi cartridges have 28 megabits of additional flash storage, directly usable by the game. This memory is organized in 4 KiB sectors, and supports the following operations:

- Identify: call `mw_flash_id_get()` to obtain the flash memory identifiers. Usually not needed.
- Erase: call `mw_flash_sector_erase()` to erase an entire 4 KiB sector. Erased sectors will be read as 0xFF.
- Program: call `mw_flash_write()` to write the specified data buffer to the indicated address. Prior to programming, **make sure the programmed address range is previously erased**, otherwise operation will fail.
- Read: call `mw_flash_read()` to read the specified amount of data from the indicated address.

This functions can be used e.g. for highscore keeping or DLCs. When using these functions, you have to keep in mind that flash can only be erased in a 1 sector (i.e. 4 KiB) granularity, and thus if e.g. you want to keep highscores, to update one of the high scores, you will have to erase the complete sector, and write it again in its entirety.

Also keep in mind that flash memory suffers from wearing, so do not perform more writes than necessary.

## 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:

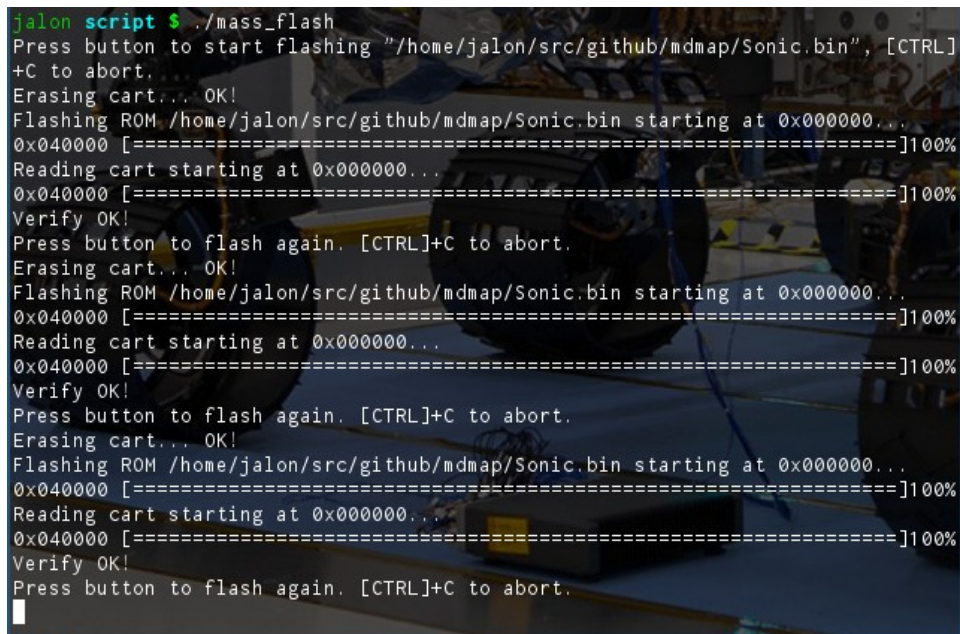
```
#!/bin/sh
FLASH=$HOME/src/github/mdmap/Sonic.bin
MDMA=$HOME/src/github/mdmap/mdma

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 17.



```
jalon script $ ./mass_flash
Press button to start flashing "/home/jalon/src/github/mdmap/Sonic.bin", [CTRL]
+C to abort.
Erasing cart... OK!
Flashing ROM /home/jalon/src/github/mdmap/Sonic.bin starting at 0x000000...
0x040000 [=====]100%
Reading cart starting at 0x000000...
0x040000 [=====]100%
Verify OK!
Press button to flash again. [CTRL]+C to abort.
Erasing cart... OK!
Flashing ROM /home/jalon/src/github/mdmap/Sonic.bin starting at 0x000000...
0x040000 [=====]100%
Reading cart starting at 0x000000...
0x040000 [=====]100%
Verify OK!
Press button to flash again. [CTRL]+C to abort.
Erasing cart... OK!
Flashing ROM /home/jalon/src/github/mdmap/Sonic.bin starting at 0x000000...
0x040000 [=====]100%
Reading cart starting at 0x000000...
0x040000 [=====]100%
Verify OK!
Press button to flash again. [CTRL]+C to abort.
```

Figure 17: mass-flash script invocation.