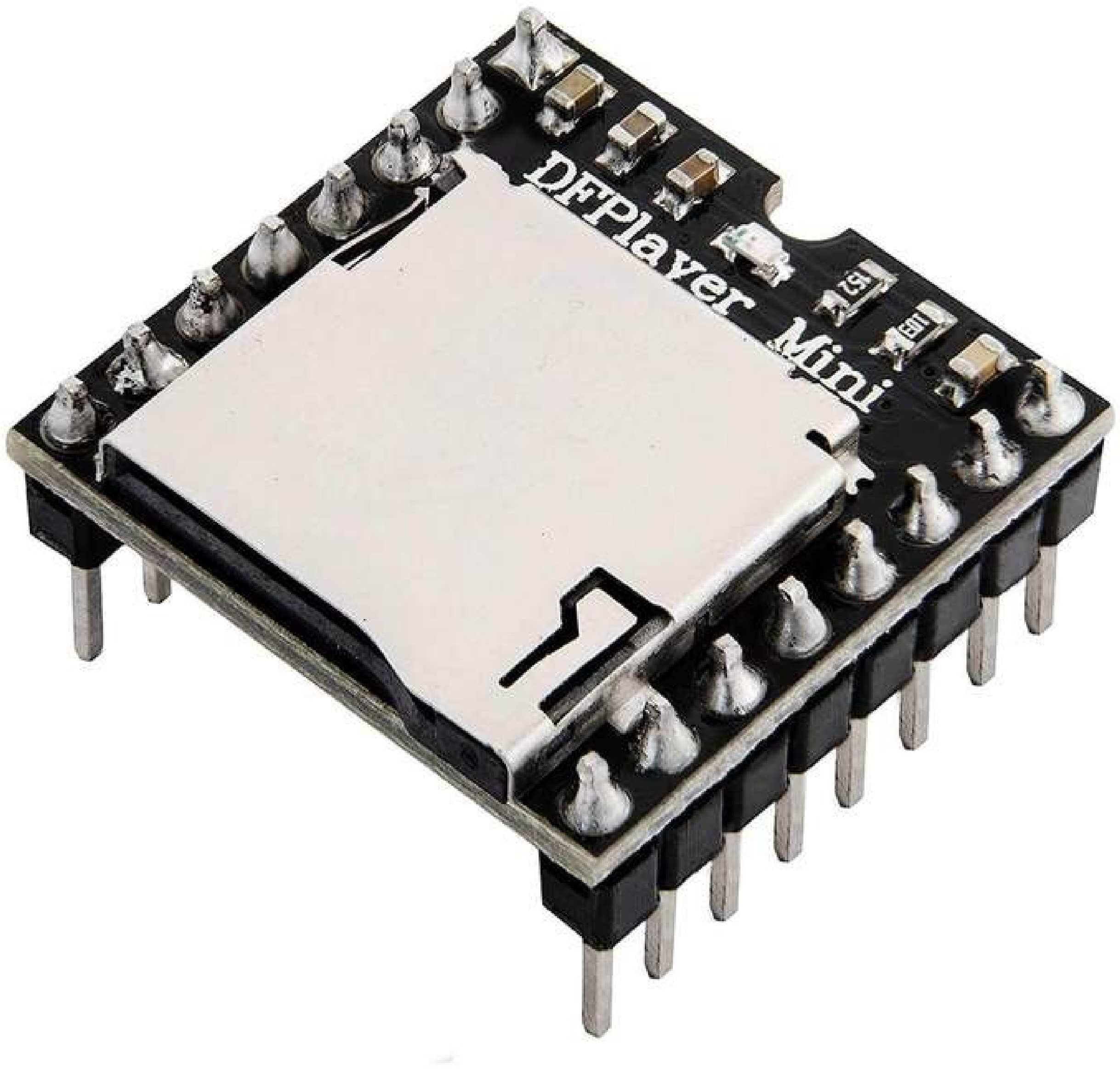
Welcome!

Thank you for purchasing our *AZ-Delivery MP3 DFPlayer Mini Module*. On the following pages, we will introduce you to how to use and set-up this handy device.

Have fun!



The MP3 DFPlayer Mini Module is a small and affordable MP3 module with output directly to the speaker or headphones. The module can be used as a stand alone module with attached battery, speaker and push buttons or used in combination with any Arduino board or any other board with USART capabilities.

The module supports common audio formats such as *MP3*, *WAV* and *WMA*. Also, it supports TF card with *FAT16*, *FAT32* file system. You can play music through a simple serial port without any complex operations.

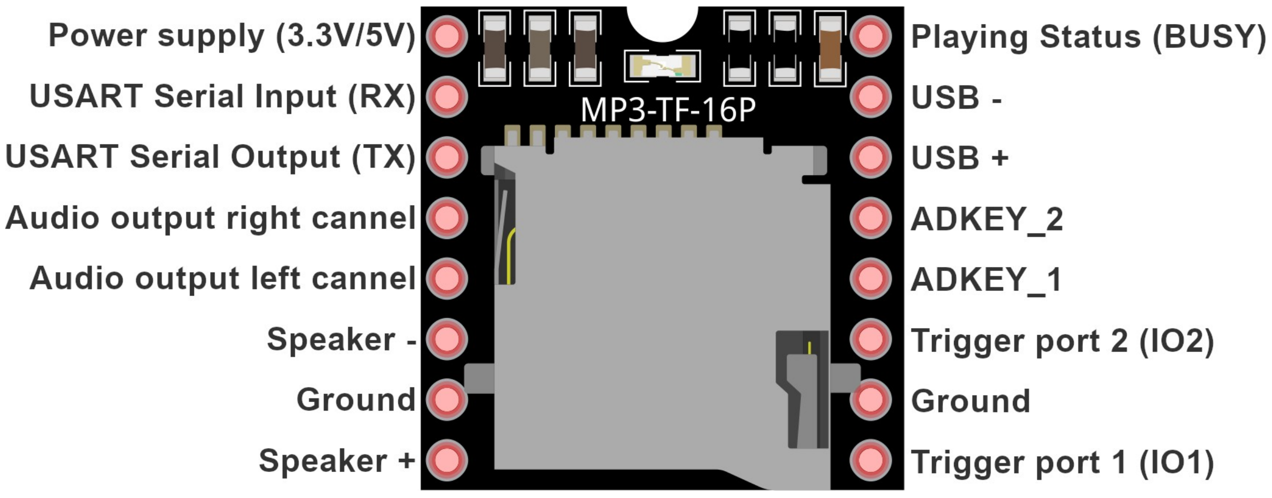
The module already comes with two *8* pin male headers pre-soldered, and it has a microSD card slot on-board. There is also a red LED on-board and it is used to indicate playing status of тхе songs. LED is connected to the *BUSY* pin of the module. *ON* state of the LED indicate that song is being played.

Specification:

»	Operating voltage range:	from 3.2V to 5V DC
»	Standby current:	20mA
»	Operating temperature:	from -40℃ to 70℃
»	UART port:	Standard serial (TTL level)
»	Baud rate:	Adjustable (default 9.600)
»	Equalizer:	6 levels, adjustable
»	Volume levels:	30 levels, adjustable
»	Sampling rates (kHz):	8/11.025/12/16/22/24/32/44.1/48
»	Output:	- 24 bit DAC range 90dB, - SNR support 85dB
»	Output power:	3W
»	Speaker resistance:	3Ω (maximum 4Ω)
»	File system:	FAT16 or FAT32
»	Maximum support:	- 32GB of the TF card, - 32GB of USB flash disk, - 64MB bytes NORFLASH
»	Control modes:	- I/O control mode, - Serial mode, - AD button control mode

- » Advertising sound waiting function (the music can be suspended; when advertising is over the music continues to play)
- » Audio data sorted by folders. The module supports up to 100 folders and every folder can hold up to 255 songs

The pinout



USART pins are used for serial communication. If you are experiencing high noise, connect one *1k*Ω resisor to the TX pin, serially.

Audio output channel pins are used as DAC pins (Digital to Analog Converter), and you should connect them to the external amplifier.

Speaker pins are pins from on-board *3W* amplifier, and you can connect them directly to the external speaker (*8*Ω max).

ADKEY pins are used for AD control mode.

Trigger port pins are used for adjusting volume levels or for switching songs (hardware). These pins are active *LOW*.

USB pins are used for connecting to the USB Flash memory stick.

Power supply and BUSY pin

The module supports operating voltage range from *3.2V* up to *5V DC*. Connect external power supply between Power supply pin and Ground pin.

The module serial port *TTL* logic level voltage is *3.3V*, so when you use *5V* levels (for example Uno board), connect serially a resistor with more than *1k*Ω resistance to the *RX* pin of the module. If you do not use this resistor you will experience high noise on the audio (speaker) output channels.

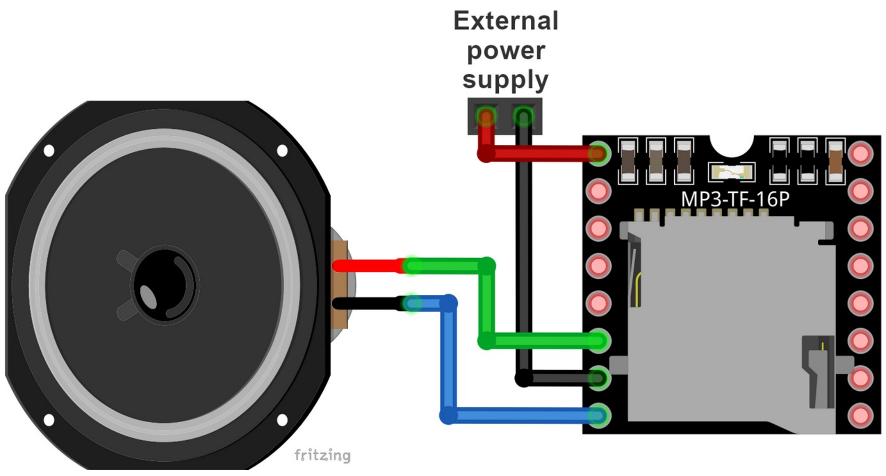
Playing status or *BUSY* pin is used to indicate playing status of a song. *LOW* state on this pin indicates that the module is currently playing a song, and *HIGH* state indicates that the module is not playing any song.

Audio output channels and speaker pins

The main chip of the module has *24* bit digital to analog converter (DAC for short). Chip has two DAC pins which are connected directly to the audio output channels of the module. You should connect external amplifier on the audio output channel pins in order to use DAC capabilities of the module. If you want to use these pins, first enable DAC output by sending corresponding command to the chip (later in the text).

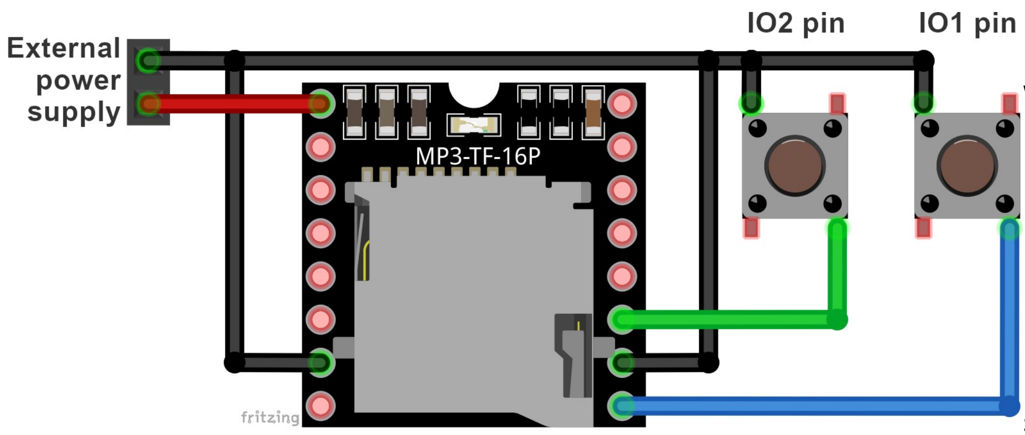
There is *3W* amplifying circuit on-board the module. The heart of this circuit is a device called "*8002 audio amplifier*", an *8* pin integrated circuit (IC) which output is connected to the `Speaker+` and `Speaker-` pins. The output of *8002* IC is mono sound.

Connect the external speaker to the module as shown on the connection diagram below:



Trigger ports (IO pins)

These pins are used for song switching and adjusting volume levels via hardware. Connect push buttons to these pins as shown on the connection diagram:



When push button is in rest state, diagonal pins of the push button are not connected. When you press the push button diagonal pins of the push button are connected, which then puts the push button in an active state.

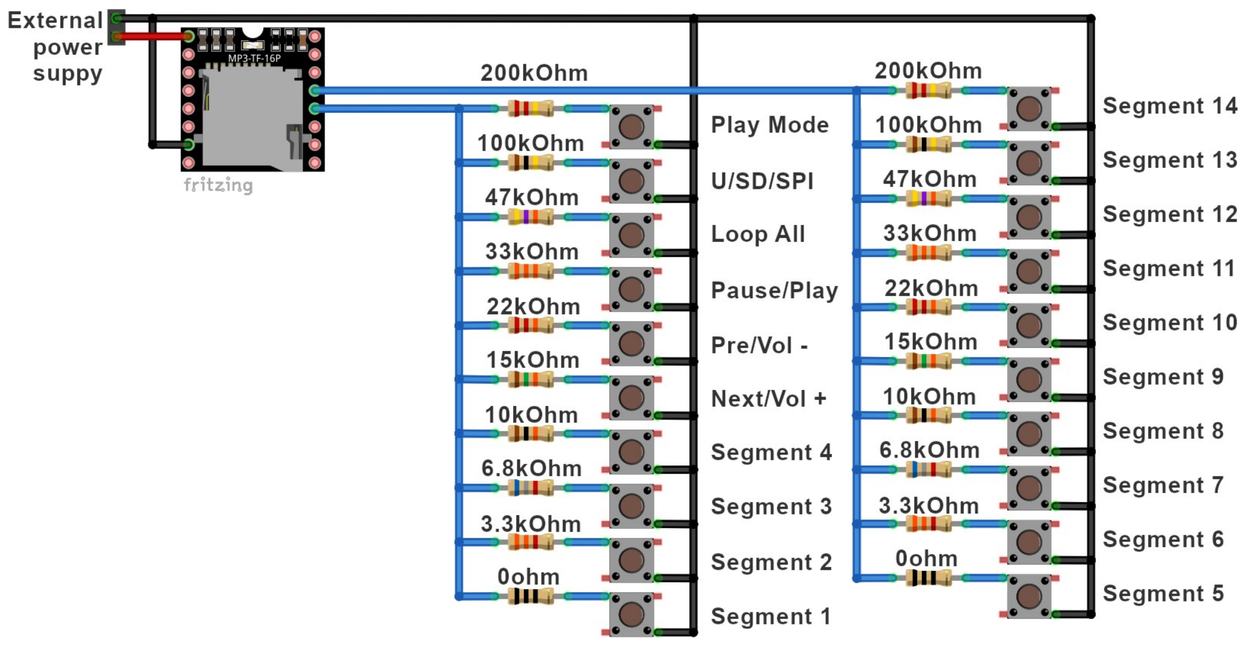
Short push on the *IO2* push button plays next song. Long push on the *IO2* push button increases the volume level.

Short push on the *IO1* push button plays previous song. Long push on the *IO2* push button decreases the volume level.

Length of the long push is more than a second. Anything less than a second is considered short push.

ADKEY pins

The AD functionality of the chip on-board the module enables you to connect *20* resistors with buttons on two AD ports of the module as shown on the connection diagram below:



NOTE: It is necessary that the power supply is as stable as possible in order for this to work properly!

Short push on the *Play Mode* button switches the playback to *interrupted* or *not interrupted*. This means that the playback will or will not be interrupted with advertisement. There is no long push function for this button.

Short push on the *U/TF/SPI* button switches the playback device to one of the following *U* = USB flash disk, *TF* = SD card, *SPI* = NORFLASH or *Sleep*. There is no long push function for this button.

Short push on the *`Loop All`* button switches the play mode to *loop all* or *not looping* of all songs. There is no long push function for this button.

Short push on the *`Pause/Play`* button pauses or plays currently selected song. There is no long push function for this button.

Short push on the *`Pre/Vol+`* button plays the previous song. Long push on the *`Pre/Vol+`* button increases the volume level.

Short push on the *`Next/Vol-`* button plays the next song. Long push on the *`Pre/Vol+`* button decreases the volume level.

Short push on the *`Segment1`* button plays the song number 1. Long push on the *`Segment1`* button repeats playing the same song.

Functions are the same for all other *`Segment`* buttons, except the song number which differs.

Serial port

RX and *TX* pins are used to establish serial communication with external microcontroller. Do not forget to connect a resistor to the *RX* pin when using *5V TTL* logic. Serial port of the module supports asynchronous serial communication mode. Default baud rate of the serial communication is *9600bps* and it is adjustable in software.

You can use serial port to send simple commands to the module and therefore control many functions that the module supports. More about commands in the next chapter.

Specification

Default baud rate:	9600bps
Data bits:	1
Checkout:	none
Flow control:	none

Format of the command

To send a command to the module, follow specific format:

`$SB VB LB CMD ACK DATA1 DATA2 CHKS1 CHKS2 $EB`

Mark	Byte	Byte description
`$SB`	*`0x7E`*	Start byte
`VB`	*`0xFF`*	Version byte
`LB`	*`0xxx`*	The number of bytes of the command without start and end bytes (In our case *`0x06`*)
`CMD`	*`0xxx`*	Such as *PLAY* and *PAUSE* and so on
`ACK`	*`0xxx`*	Acknowledge byte *`0x00`* = not ack, *`0x01`* = ack
`DATA1`	*`0xxx`*	Data high byte
`DATA2`	*`0xxx`*	Data low byte
`CHKS1`	*`0xxx`*	Checksum high byte
`CHKS2`	*`0xxx`*	Checksum low byte
`$EB`	*`0xEF`*	End byte

Acknowledge byte is used to get data from the module. If it is set to *`0x00`* no data will be sent from the module and if it is set to *`0x01`* you will get response data from the module. The length of the data is not limited but usually it has two bytes (*`data1`* and *`data2`* bytes).

To send a specific command, just send byte by byte serially over software serial interface (you can see this later in the code).

Folder structure and song names

The module supports several types of folders and specific names for songs. Names of folders are numbers, except "*mp3*" and "*ADVERT*" folders. Song names have to start with a number after which comes the string without spaces. Example of the song name:
`0001_Linking_Park_In_The_End.mp3`

There are *5* types of folders.

First is a type of folder that can contain *256* songs. The module supports *256* of these folders in total. The names of these folders are numbers in the range from *000* to *255*. Song names in these folders start with numbers in the range from *000* to *255*. The module does not support subfolders in these folders.

Second is a type of folder that can contain *3000* songs. The module supports *16* of these folders in total. The names of these folders are numbers in the range from *00* to *15*. Song names in these folders start with numbers in the range from *0000* to *2999*. The module does not support subfolders in these folders.

Third is a folder called "*mp3*" and it too can contain *3000* songs. The module supports one "*mp3*" folder in total. Song names in this folder start with numbers in the range from *0000* to *2999*. The module does not support subfolders in this folder.

Fourth is a folder called "*ADVERT*" and it too can contain *3000* songs and it is used for advertisement songs. The module supports one "*ADVERT*" folder in total. Song names in this folder start with numbers in the range from *0000* to *2999*. The module does not support subfolders in this folder.

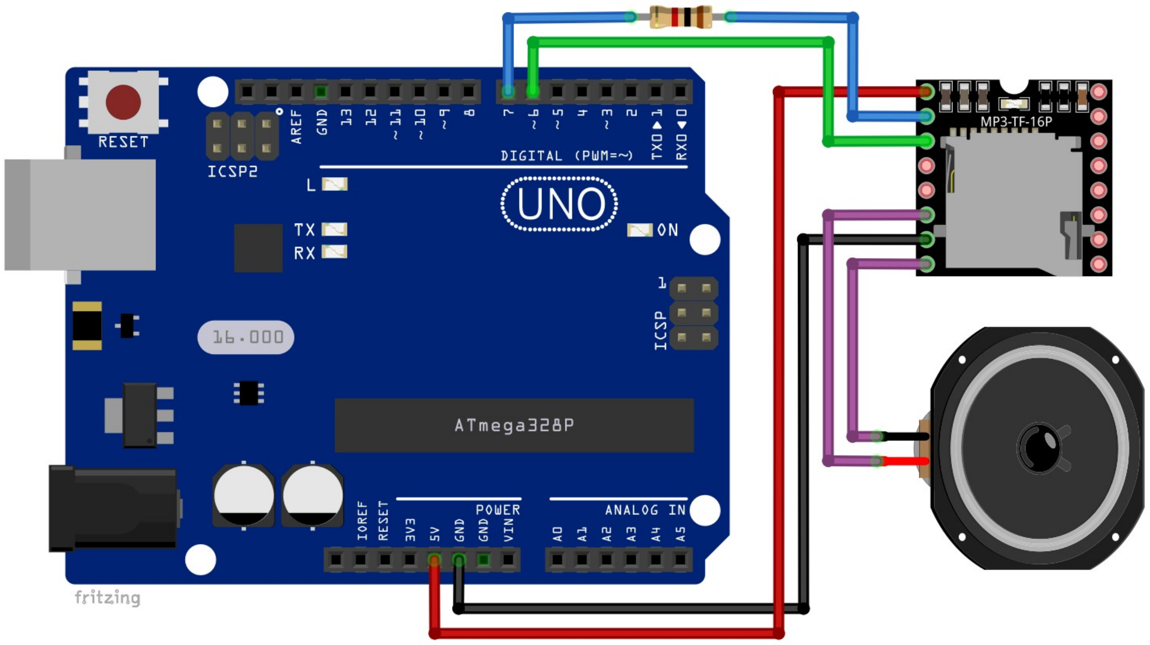
And, the fifth folder type is "*root*" folder. If this is the only folder on SD card, or USB memory, (no other folders) this folder can contain up to *65536* songs. This folder can have subfolders, including any or all folder types. The "*root*" folder can contain songs and subfolders at the same time.

Example of folder structure:

```
root
--- 0001r.mp3
--- 0002r.mp3
--- 0003r.mp3
--- 0004r.mp3
--- 0001
--- --- 0001x.mp3
--- --- 0002x.mp3
--- 0002
--- --- 0001y.mp3
--- mp3
--- --- 0001m.mp3
--- --- 0002m.mp3
--- --- 0003m.mp3
--- ADVERT
--- --- 0001a.mp3
--- --- 0002a.mp3
```

Sending commands to the module

In order to send commands to the module, connect the module with the Uno as shown on the connection diagram below:



Module pin	>	Uno pin	
VCC	>	5V	Red wire
RX	>	D7 (via 1kΩ resistor)	Blue wire
TX	>	D6	Green wire
GND	>	GND	Black wire

Module pin	>	Speaker pin	
SPK -	>	One side of the speaker	Purple wire
SPK+	>	The other side of the speaker	Purple wire

NOTE: You can use any other board with a microcontroller which has USART capabilities.

We are using serial interface created in software on digital I/O pins *6* and *7* of the Uno, because Uno uses hardware serial pins (digital I/O pins *0* and *1*), for programming main microcontroller.

The microcontroller can not send commands to control the module until initialization of the module is finished and data is returned. Otherwise the commands sent by microcontroller will be ignored and also this will effect the initialization process.

If not stated otherwise (by sending a command after initialization), when the module is powered *ON*, it reads SD card first and if SD card is not available it switches to USB flash disk.

Sketch example:

```
#include "SoftwareSerial.h"
#define Start_Byte        0x7E
#define Version_Byte      0xFF
#define Command_Length    0x06
#define End_Byte          0xEF
// Returns info with command 0x41 [0x01: info, 0x00: no info]
#define Acknowledge       0x01
SoftwareSerial mySerial(6, 7);  //  RX, TX
byte receive_buffer[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
char data;  // Used for received commands from Serial Monitor
byte volume = 0x00;       // Used to store current volume level
bool mute_state = false;  // Used to toggle mute state

// Excecute the command and parameters
void execute_CMD(byte Command, byte Data1, byte Data2) {
  // Calculate the checksum (2 bytes)
  word Checksum = -( Version_Byte + Command_Length + Command +
                                 Acknowledge + Data1 + Data2);
  // Build the command
  byte command_line[10] = { Start_Byte, Version_Byte,
                            Command_Length, Command, Acknowledge,
                            Data1, Data2, highByte(Checksum),
                            lowByte(Checksum), End_Byte};
  //  Send the command line to the module
  for(byte k = 0; k < 10; k++) {
    mySerial.write(command_line[k]);
  }
}
```

```
void reset_rec_buf() {
  for(uint8_t i = 0; i < 10; i++) {
    receive_buffer[i] = 0;
  }
}

bool receive() {
  reset_rec_buf();
  if(mySerial.available() < 10) {
    return false;
  }
  for(uint8_t i = 0; i < 10; i++) {
    short b = mySerial.read();
    if(b == -1) {
      return false;
    }
    receive_buffer[i] = b;
  }
  // When you reset the module in software,
  // received buffer elements are shifted.
  // To correct that we do the following:
  short b = receive_buffer[0];
  for(uint8_t i = 0; i < 10; i++) {
    if(i == 9) {
      receive_buffer[i] = b;
    }
    else {
      receive_buffer[i] = receive_buffer[i+1];
    }
  } // End correcting receive_buffer
  return true;
}
```

```
void print_received(bool print_it) {
  if(print_it) {
    if(receive()) {
      for(uint8_t i = 0; i < 10; i++) {
        Serial.print(receive_buffer[i], HEX); Serial.print("\t");
      }
      Serial.println();
    }
  }
  else { receive(); }
}

void module_init() {
  execute_CMD(0x0C, 0, 0); delay(1000); // Reset the module
  print_received(false);   delay(100);
  Serial.print("SDON\t");
  print_received(true);    delay(100);
  playFirst();
  setVolume(0x09);
}

void play_first() {
  Serial.print("PLYFST\t");
  execute_CMD(0x03, 0, 1); delay(100); // Play first song
  print_received(false);   delay(100);
  execute_CMD(0x45, 0, 0); delay(100); // Get playback status
  print_received(false);   delay(100);
  print_received(true);    delay(100);
}
```

```
void set_volume(uint8_t volume) {
  Serial.print("SETVOL\t");
  execute_CMD(0x06, 0, volume); delay(100); // Set volume level
  print_received(false);        delay(100);
  execute_CMD(0x43, 0, 0);      delay(100); // Get volume level
  print_received(false);        delay(100);
  print_received(true);         delay(100);
}
void play_next() {
  Serial.print("NEXT\t");
  execute_CMD(0x01, 0, 0); delay(100);
  print_received(false);   delay(100);
  execute_CMD(0x4C, 0, 0); delay(100); // Get current song played
  print_received(false);   delay(100);
  print_received(true);    delay(100);
}
void mute() {
  mute_state = !mute_state;
  if(mute_state) {
    execute_CMD(0x43, 0, 0); delay(100); // Return volume leve
    print_received(false);   delay(100);
    print_received(false);   delay(100);
    volume = receive_buffer[6];
    Serial.print("MUTE\t");
    execute_CMD(0x06, 0, 0x00); delay(100); // Set volume level
    print_received(false);      delay(100);
    execute_CMD(0x43, 0, 0);    delay(100); // Get volume level
    print_received(false);      delay(100);
    print_received(true);       delay(100);
  }
// one tab
  else {
    Serial.print("VOL\t");
    execute_CMD(0x06, 0, volume); delay(100); // Set previous vol
    print_received(false);        delay(100);
    execute_CMD(0x43, 0, 0);      delay(100); // Get volume level
    print_received(false);        delay(100);
    print_received(true);         delay(100);
  }
}

void random_play() {
  // Random plays all songs, loops all, repeats songs in playback
  execute_CMD(0x18, 0, 0);
  delay(100);
  Serial.print("RNDM\t");
  print_received(false);
  delay(100);
  execute_CMD(0x4C, 0, 0); // Get current song played
  delay(100);
  print_received(false);
  delay(100);
  print_received(true);
  delay(100);
}
```

```
void query_status() {
  execute_CMD(0x42, 0, 0); delay(100); // Get status of module
  print_received(false);   delay(100);
  Serial.print("STATUS\t");
  print_received(true);    delay(100);
  execute_CMD(0x43, 0, 0); delay(100); // Get volume level
  print_received(false);   delay(100);
  Serial.print("VOLUME\t");
  print_received(true);    delay(100);
  execute_CMD(0x44, 0, 0); delay(100); // Get EQ status
  print_received(false);   delay(100);
  Serial.print("EQ\t");
  print_received(true);    delay(100);
  execute_CMD(0x45, 0, 0); delay(100); // Get playback status
  print_received(false);   delay(100);
  Serial.print("PLYBCK\t");
  print_received(true);    delay(100);
  execute_CMD(0x46, 0, 0); delay(100); // Get software version
  print_received(false);   delay(100);
  Serial.print("SFVER\t");
  print_received(true);    delay(100);
  // Get total number of files on storage device
  execute_CMD(0x48, 0, 0); delay(100);
  print_received(false);   delay(100);
  Serial.print("FILES\t");
  print_received(true);    delay(100);
  execute_CMD(0x4C, 0, 0); delay(100); // Get current song played
  print_received(false);   delay(100);
  Serial.print("CRRTRK\t");
  print_received(true);    delay(100);
}
```

```
void setup() {
  Serial.begin(115200);
  mySerial.begin(9600);  delay(1000);

  Serial.println("\nInitialization");
  module_init();
}
void loop() {
  print_received(true);

  while(Serial.available() > 0) {
    data = Serial.read();
    // Serial.println(data, HEX); // For debugging
    if(data != "/n") {
      if(data == 'N') {
        Serial.println("\nPlay next song");
        play_next();
      }
      else if(data == 'B') {
        Serial.println("\nRandom play");
        random_play();
      }
      //  .....
      else if(data == 'D') {
        Serial.println("\nQuerry status of the module");
        query_status();
      }
    }
  }
  delay(100);
}
```

The sketch starts with including one library called “*SoftwareSerial.h*”.

Then we define five macros. These macros represents the command bytes that are the same for all commands. First byte is called ”*Start_Byte*” which value is *0x7E*, second byte is called “*Version_Byte*” which value is *0xFF*, third byte is called “*Command_Length*” which value is *0x06*, fourth byte is called “*End_Byte*” which value is *0xEF* and fifth byte is called “*Acknowledge*” with value *0x01*.

Then we instantiate software serial object called “*mySerial*” with this line of code: `SoftwareSerial mySerial(6, 7);`
Where *6* represent digital I/0 pin of Uno on which *RX* pin of the module is connected and *7* represent digital I/0 pin of Uno on which *TX* pin of the module is connected.

Then we create an array called “*receive_buffer*”, which has ten elements. Elements of the “*receive_buffer*” array represents bytes that are sent from the module and received by Uno.

After this we create three variables. First is called “*data*” and it is used to store commands when we send them from Serial Monitor. Second variable is called “*volume*” and it is used to store current volume level, when we send command “*Mute*”. Third variable is called “*mute_state*” and it is used to toggle mute state of the module.

Then we create several functions. First function is called "*execute_CMD()*" which accepts three arguments and returns no value. The function *execute_CMD()* is used to send commands to the module. First argument is the command byte, second is *data1* byte and third is *data2* byte of the command. At the beginning of the *execute_CMD()* function we calculate checksum bytes with this line of the code:

```
word checksum = -(Version_Byte + Command_Length + CMD +
                  Acknowledge + Par1 + Par2);
```

Then we create command array, called "*command_line*". This array has ten elements, which represents ten bytes of the command: *Start_Byte*, *Version_Byte*, *Command_Length*, *Command*, *Acknowledge*, *Data1*, *Data2*, *highByte(Checksum)*, *lowByte(Checksum)*, and *End_Byte*. At the end of *execute_CMD()* function, we use *for* loop to send all ten bytes, one by one, to the module via software serial.

Next function is called "*reset_rec_buf()*" and it is used to set all values of elements in *receive_buffer* to the zero value, or to reset the buffer. The *reset_rec_buf()* function accepts no arguments and returns no value.

Then we create a function called "*receive()*" and it is used to receive bytes from the module and store them in the *receive_buffer* array. The *receive()* function accepts no arguments and returns a boolean value. At the beginning of the *receive()* function, we call *reset_rec_buf()* to reset the *receive_buffer* array. Then we check if there is data on software serial, and if that data has ten bytes.

If the return data has ten bytes, then we use *for* loop to read all ten bytes. After reading a byte, we check if its value is valid by checking if it is different from "*-1*". If it is different, store its value to the *receive_buffer*. If any of the checks are not satisfied, returned boolean value is "*false*"; otherwise, this value is "*true*". When we reset the module in software, *receive_buffer* elements are shifted, so we need to correct that. We do the correction with the following code:

```
short b = receive_buffer[0];
for(uint8_t i = 0; i < 10; i++) {
  if(i == 9) {
    receive_buffer[i] = b;
  }
  else {
    receive_buffer[i] = receive_buffer[i+1];
  }
}
```

After this, we create a function called "*print_received()*" which is used to print received data to the Serial Monitor. The *print_received()* function accepts one argument, a boolean value which is used when determining if the data should be printed or not. If the argument value is equal to "*true*", we call *receive()* function and print out the data from *receive_buffer*. If the argument value is equal to "*false*", then we only call the *receive()* function without printing the data.

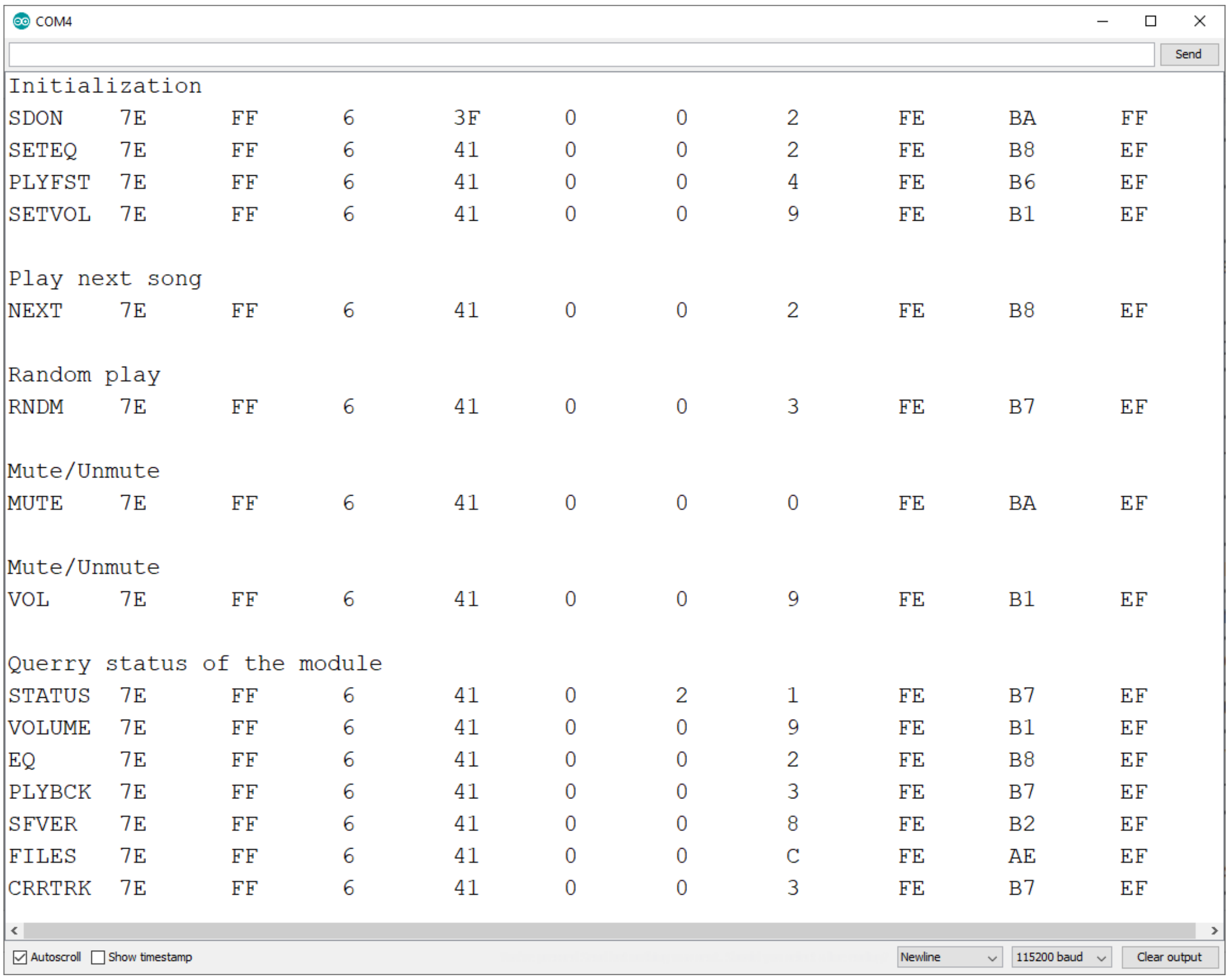
After these functions, we create several other functions that use previous functions. All of these new functions are self explanatory.

In the *`setup()`* function we start hardware serial with baud rate of *115.200* bps, and software serial with baud rate *9.600* bps (which is default baud rate of the module). Then we call the function *`module_init()`* which initializes the module, sets equalizer, volume level, plays the first song on the storage device and prints out the status data to the Serial Monitor.

In the loop() function we wait for data on the hardware serial. This data is sent from Serial Monitor when we send a command. The data is one of the following letters: N, B, D and several other letters. We check which letter is sent and then call corresponding function.

The sketch code in this eBook is just an example, a part from our sketch example. If you want to see complete sketch, visit the repository on the following *GitHub* link:
https://github.com/Slaveche90/DFPlayer_Custom_Sketch

When you upload the complete sketch example to the Uno, start the Serial Monitor (*Tools* > *Serial Monitor*), and send few letters from the sketch via Serial Monitor to the Uno. The output should look like the output on the image below:



```
Initialization
SDON     7E       FF       6        3F       0        0        2        FE       BA       FF
SETEQ    7E       FF       6        41       0        0        2        FE       B8       EF
PLYFST   7E       FF       6        41       0        0        4        FE       B6       EF
SETVOL   7E       FF       6        41       0        0        9        FE       B1       EF

Play next song
NEXT     7E       FF       6        41       0        0        2        FE       B8       EF

Random play
RNDM     7E       FF       6        41       0        0        3        FE       B7       EF

Mute/Unmute
MUTE     7E       FF       6        41       0        0        0        FE       BA       EF

Mute/Unmute
VOL      7E       FF       6        41       0        0        9        FE       B1       EF

Querry status of the module
STATUS   7E       FF       6        41       0        2        1        FE       B7       EF
VOLUME   7E       FF       6        41       0        0        9        FE       B1       EF
EQ       7E       FF       6        41       0        0        2        FE       B8       EF
PLYBCK   7E       FF       6        41       0        0        3        FE       B7       EF
SFVER    7E       FF       6        41       0        0        8        FE       B2       EF
FILES    7E       FF       6        41       0        0        C        FE       AE       EF
CRRTRK   7E       FF       6        41       0        0        3        FE       B7       EF
```

AZ-Delivery

Command examples

Command	Bytes (HEX) *	Description
Next Song	7E FF 06 **01** 00 00 00 EF	Play next song
Previous Song	7E FF 06 **02** 00 00 00 EF	Play previous song
Play with index	7E FF 06 **03** 00 **00 01** EF	Play the first song
	7E FF 06 **03** 00 **00 02** EF	Play the second song
Volume up	7E FF 06 **04** 00 00 00 EF	Volume increase one level
Volume down	7E FF 06 **05** 00 00 00 EF	Volume decrease one level
Set volume	7E FF 06 **06** 00 **00 1E** EF	Set the volume to 30 (=0x1E)
Set EQ	7E FF 06 **07** 00 00 **02** EF	Set EQ to 02 – Rock; 00 / 01 / 02 / 03 / 04 / 05 Normal/Pop/Rock/Jazz/Classic/Base
Loop specific song	7E FF 06 **08** 00 **00 01** EF	Loop song 0001
Select device	7E FF 06 **09** 00 00 **01** EF	Select storage device to USB memeory
	7E FF 06 **09** 00 00 **02** EF	Select storage device to SD card
Sleep mode	7E FF 06 **0A** 00 00 00 EF	Chip enters sleep mode
Wake up	7E FF 06 **0B** 00 00 00 EF	Chip wakes up
Reset	7E FF 06 **0C** 00 00 00 EF	Chip reset
Play	7E FF 06 **0D** 00 00 00 EF	Resume the playback
Pause	7E FF 06 **0E** 00 00 00 EF	Playback is paused
Play specific song in a folder that supports 256 songs; module suports 256 folders (0 - 255) with 255 songs.	7E FF 06 **0F** 00 **01 01** EF	Play the song with the folder: 01/0001xxx.mp3
	7E FF 06 **0F** 00 **01 02** EF	Play the song in the folder: 01/0002xxx.mp3
Audio amplification	7E FF 06 **10** 00 **01 0A** EF	01 – Amp ON; 0A – level (0-31)
	7E FF 06 **10** 00 **00 00** EF	00 – Amp OFF
Loop all	7E FF 06 **11** 00 00 **01** EF	Start loop all songs
	7E FF 06 **11** 00 00 **00** EF	Stop looping all songs and stop playback
Play in mp3 folder	7E FF 06 **12** 00 **00 01** EF	Play song 0001 in mp3 folder (0x0001 – 0x0BB8; 3000 songs)
Play an add	7E FF 06 **13** 00 **00 01** EF	Play the song 0001 in folder ADVERT (0x0001 – 0x0BB8; 3000 songs)

*** Command bytes without two checksum bytes**

Command	Bytes (HEX) *	Description
Play specific song in a folder that supports 3000 songs; module suports 16 folders (0 - 15) with 3000 songs.	7E FF 06 **14** 00 **00 01** EF	In folder 0 play song 001
	7E FF 06 **14** 00 **91 11** EF	In folder 9 play song 273 (=0x111)
	7E FF 06 **14** 00 **F0 05** EF	In folder 15 (=0xF) play song 005
Stop playing add	7E FF 06 **15** 00 00 00 EF	Stop playing adverstment and resume previous playback
Enable loop all	7E FF 06 **16** 00 00 **01** EF	Enable loop all and start playing song 1
Stop play	7E FF 06 **16** 00 00 **00** EF	Stop the playback
Loop song in folder that supports 256 songs	7E FF 06 **17** 00 **01 02** EF	Loop song 02 in the 01 folder
Random playback	7E FF 06 **18** 00 00 00 EF	Random play all songs on the device
Set single loop play	7E FF 06 **19** 00 00 **00** EF	Start current song loop play
	7E FF 06 **19** 00 00 **01** EF	Stop current song loop play
Set DAC	7E FF 06 **1A** 00 00 **00** EF	Start DAC output
	7E FF 06 **1A** 00 00 **01** EF	Stop DAC output
Play specific song with volume	7E FF 06 **22** 00 **1E 01** EF	Set the volume to 30 (0x1E is 30) and play the first song
	7E FF 06 **22** 00 **0F 02** EF	Set the volume to 15 (0x0F is 15) and play the second song

*** Command bytes without two checksum bytes**

Status updates of the module

There is an option if you want to get the return data from the module. This data is very useful because it can contain information of current playback status, volume level, EQ option, when the current playing song is finished, etc. To enable this option you have to set the `Acknowledge` byte of the command to the value *0x01* (*0x00* no return data).

When you send command with `Acknowledge` byte set to *0x01*, data returns. Here is the list of commands to be send to the module in oreder to get status updates:

Command bytes (HEX) *	Description
7E FF 06 **3F** 00 00 00 EF	To get current storage device send this command
7E FF 06 **40** 00 00 **01** EF	This is return data, and it indicates error, where 01 is error value
7E FF 06 **41** 00 **00 00** EF	This is return data with no error. This indicates successifully received and executed command, where 00 00 is status of the module
7E FF 06 **42** 00 00 00 EF	To get playback status send this command
7E FF 06 **43** 00 00 00 EF	To get current volume level send this command
7E FF 06 **44** 00 00 00 EF	To get current EQ status send this command
7E FF 06 **47** 00 00 00 EF	To get total number of files on USB flash disk send this command
7E FF 06 **48** 00 00 00 EF	To get total number of files on SD card send this command
7E FF 06 **4B** 00 00 00 EF	To get current song number on USB flash disk send this command
7E FF 06 **4C** 00 00 00 EF	To get current song number on SD card send this command
7E FF 06 **4E** 00 00 00 EF	To get total number of files on any storage media send this command
7E FF 06 **4E** 00 00 **02** EF	To get total number of files in the folder 02 send this command
7E FF 06 **4E** 00 00 **0C** EF	To get total number of files in the folder 12 send this command
7E FF 06 **4F** 00 00 00 EF	To get total number of folders on any storage device send this command

*** Command bytes without two checksum bytes**

Return values

The return data is in format:

`0x7E 0xFF 0x06` **`0x41`** `0x00` **`A  B`** `checksum1 checksum0 0xEF`

The value ***0x41*** indicates that a ommand was received by the module and executed successfully.

The value “***A***” represents storage media, where:

A = *0x01* - USB flash disk, and

A = *0x02* - SD card.

The value "***B***" indicates status of the playback, where

B = *0x00* indicates that the playback is stopped,

B = *0x01* indicates that the playback is playing and

B = *0x02* indicates that the playback is paused.

Example of returned data:

`0x7E 0xFF 0x06` **`0x41`** `0x00` **`0x02 0x01`** `0xFE 0xF7 0xEF`

where:

`0x02` – storage device is SD card

`0x01` – playback is currently playing

Errors

If some error occures, the return data will be in the format:

`0x7E 0xFF 0x06` **`0x40`** `0x00 0x00` **`0x01`** `chks1 chks0 0xEF`

Where *`0x40`* indicates that error occurred, and *`0x01`* indicates error value.

Error values with descriptions are in the table below:

Error data (HEX) *	Description
7E FF 06 **40** 00 00 **01** EF	The module is busy
7E FF 06 **40** 00 00 **02** EF	The module is in sleep mode
7E FF 06 **40** 00 00 **03** EF	Serial reveiving error (frame is not received completely yet)
7E FF 06 **40** 00 00 **04** EF	Checksum incorret error
7E FF 06 **40** 00 00 **05** EF	Specified song is out of current songs scope
7E FF 06 **40** 00 00 **06** EF	Specified song is not found
7E FF 06 **40** 00 00 **07** EF	Intercut error (adverstment can only be played on playing song, not paused or stopped)
7E FF 06 **40** 00 00 **08** EF	SD card reading error (SD card is demaged or pulled out)
7E FF 06 **40** 00 00 **0A** EF	The module entered sleep mode

*** Error bytes without two checksum bytes**

Specific returned data

If the acknowledge byte is set to *`0x01`*, the module will output data when song is finished, when SD card (or USB flash disk) is pushed *IN* or pulled *OUT* or when storage device is online. These values will be returned without sending any command to the module.

The returned data of storage device when it is pushed *IN*:

`0x7E 0xFF 0x06` **`0x3A`** `0x00 0x00` **`A`** `0xFE 0xF7 0xEF`

where:

`0x3A` indicates that storage device is pushed *IN*

The returned data of storage device when it is pulled *OUT*:

`0x7E 0xFF 0x06` **`0x3B`** `0x00 0x00` **`A`** `0xFE 0xF7 0xEF`

where:

`0x3B` indicates that storage device is pulled *OUT*

A = *`0x01`* indicates that storage device is USB flash disk

A = *`0x02`* indicates that storage device is SD card

A = *`0x04`* indicates that USB cable is connected or not connected to the PC

The returned data of finished song is in the following format:

`0x7E 0xFF 0x06` **`0x3D`** `0x00` **`0x00`** **`0x05`** `0xFE 0xF7 0xEF`

where:

`0x3D` indicates the song is finished on SD card (*`0x3C`* = on USB flash disk),

`0x00 0x05` indicates the song name "*`0005`*".

The returned data when storage device is online:

`0x7E 0xFF 0x06` **`0x3F`** `0x00 0x00` **`A`** `0xFE 0xF7 0xEF`

where:

`0x3F` indicates that storage device is online, and

"***A***" can have several different values:

A = *`0x01`* indicates USB flash disk

A = *`0x02`* indicates SD card

A = *`0x03`* indicates that USB flash disk and SD card are both online at the same time

A = *`0x04`* indicates PC connection

Playback returned values

If the acknowledge byte is set to *`0x01`*, we send the command for playback status:

```
0x7E 0xFF 0x06 0x45 0x00 0x00 0x00 chks1 chks0 0xEF
```

The returned data will be in format:

```
0x7E 0xFF 0x06 0x41 0x00 0x00  A  chks1 chks0 0xEF
```

where "***A***" can have several different values:

A = *`0x00`* indicates that playback is set to play and loop all songs on the storage device, one by one,

A = *`0x01`* indicates that playback is set to play and loop all songs in specific folder, one by one,

A = *`0x02`* indicates that playback is set to play and loop one song,

A = *`0x03`* indicates that playback is set to random play and loops all songs on the storage device; In random play songs will be repeated,

A = *`0x04`* indicates that playback is set to play one song and when the song is finished, playback stops.

You've done it!

Now you can use your module for various projects.

Now is the time to learn and make the Projects on your own. You can do that with the help of many example scripts and other tutorials, which you can find on the internet.

If you are looking for the high quality products for Arduino and Raspberry Pi, AZ-Delivery Vertriebs GmbH is the right company to get them from. You will be provided with numerous application examples, full installation guides, eBooks, libraries and assistance from our technical experts.