MSX Pi Interface v. 0.8.1

Ronivon Candido Costa
old@retro-cpu.run

Technical Reference Manual

Table of Contents

1 Users Guide

MSXPi boots into MSX-DOS 1 from /home/pi/msxpi/disks/msxpiboot.dsk. The boot disk has all currently available tools, and new tools are made available via a separate release process and a new tools disk named "msxpitools.dsk".

From MSX-DOS, you can use all resources available to MSX-DOS1, such as game loaders, plus a set of commands specifically developed for you to enjoy some resources from Pi (described later).

From DOS, you can jump to BASIC typing BASIC as usual.

From the BASIC prompt, type any of these commands:

```
CALL MSXPIVER
CALL MSXPISTATUS
CALL MSXPILOAD
```

1.1 MSXPi Client Commands

After starting the client with command "call msxpiload", the following is displayed in the screen.

MSXPi Hardware Interface v0.7
MSXPi Cloud OS (Client) v0.8.1
TYPE HELP for available commands
CMD:

The available commands can be viewed with the HELP command:

CMD:HELP
BASIC CHKPICONN CLS PDIR HELP PLOADBIN PLOADROM PMORE #(Pi Command) PRESET

CHKPICONN: will verify if Pi is responding. Result will be printed on screen, depending on status.
DIR: will show files in current path. Dir will work for files on the Raspberry Pi filesystem, and also remote

PLOADBIN/PLOADROM: Load a binary program into memory. Currently supported formats are binary files to run from BASIC (PLOADBIN), and .rom (PLOADROM for roms up to 32KB. The loader is very basic and cannot load all rom types.
File is loaded into the address extracted from the header, even for a ROM file. In this case, inter-slot calls are made to write the rom into page 1 (0x4000), which makes the load process slower.

#: This command passes all commands directly to the Raspberry Pi for execution. The command is executed, and the output printed in the MSX screen. It does not support input to the command, such as prompts. The command must execute and terminate, returning control to the MSXPi.
Examples of usage:
#ls
#pwd
#hostname
#unzip msxgame.zip

RESET: Sent reset command to both MSXPi interface and MSXPi Server application.

1.2 For ROM-less MSXPi

MSXPi prototypes that was released without EPROM should be used along a MegaFLashRom cartridge. The file MSXPi-DOS_16K.ROM should be written to the MFR and MSXPi will work as a regular MSXPi with integrated EPROM.

2 Architecture and Behavior Description

Following picture illustrate the high level design of the MSXPi interface.

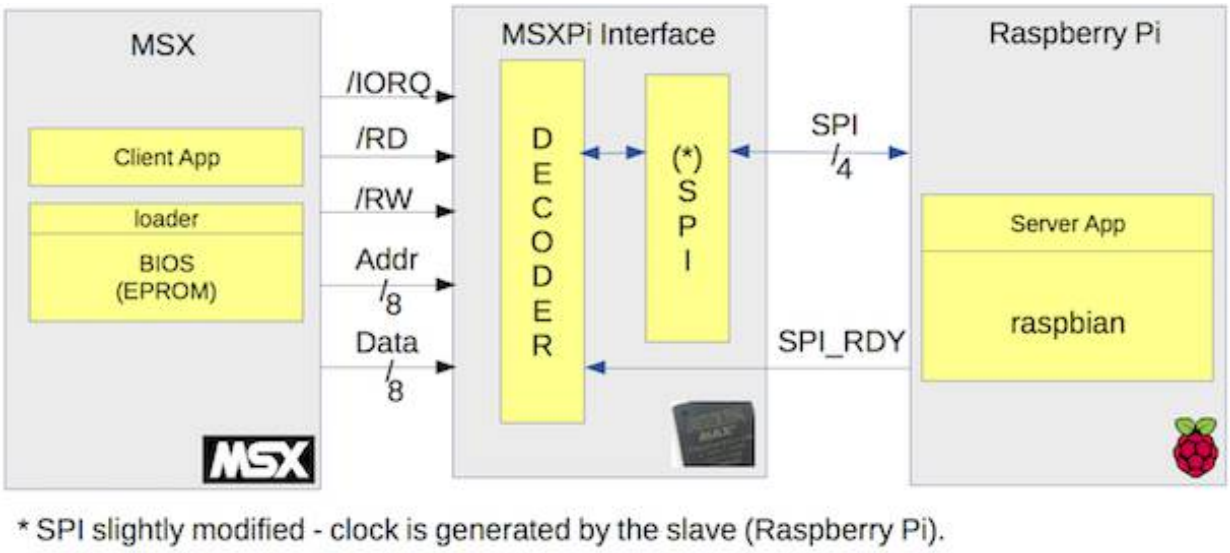


Ilustração1: Arquitetura

1. MSX starts a transfer with Pi, by one of two means:
1a. MSX send a read signal to Pi, by writing 0 to port 0x06;
1b. MSX send a write signal to Pi, by writing a value to port 0x07.
2. CPLD decodes addresses and buffers the MSX byte.
3. SPI sends a signal to Pi to start a new transfer, lowering SPI_CS.
4. Pi send back to the Interface the busy signal, lowering SPI_RDY.
5. Pi starts generating clock pulses into SPI_SCLK pin.
6. Bit shifter inside CPLD convert PI serial bits to a byte, and MSX byte to serial bits. Transfer is done in both ways (full-duplex). Pi byte is buffered into the CPLD.
7. MSX read port 0×07 to receive the byte send by Pi.

The interface uses serial transfer between CPLD and Raspberry Pi. Because of this technical feature, it is not ideal for some applications that require high throughput, such as graphics applications. Currently the benchmark transfer rate is approximately 5KB per second (Kilo Bytes / s).

The interface between MSX and Pi is made through a 5 volt CPLD tolerant, the EPM3064 from Altera. In this CPLD is implemented a logic transfer a byte between MSX and Raspberry Pi in serial full duplex mode using a variant of the SPI protocol with a small change in the generation of the clock, which is being generated by the pi (slave) instead of being Generated by MSX or by the interface itself. This change allows the Pi to specify the transfer speed.

The client application was developed in the Assembly Z80 language, and the server application in C.

On the Raspberry side, the application runs on the Raspbian, and is automatically initialized every time the Pi is connected by the systemd service.

By using a standard linux system, the solution allows a wide range of applications to be easily developed through the use of existing tools in Raspberry, both for programming and for accessing Pi GPIOs.

3 Connecting MSXPi to the Raspberry Pi

The interface is compatible with any model of Raspberry Pi, although it was developed for the "Zero" model, so that it is fully incorporated into the cartridge. The only exposed parts will be SD, USB and LED card slots.
The GPIO pins (see Appendix 1) of the Pi used by the interface are:

GPIO	Purpose	Direction	Enabled	CPLD Pin
21	/CS (SPI Chip Select)	Input	0	28
20	SCLK (SPI Clock)	Output	N/A	34
16	MOSI (Interface out)	Input	N/A	5
12	MISO (Interface in)	Output	N/A	41
25	PI_Ready	Output	1	12
	GND (Pin 4)	N/A	N/A	
	VCC (Pin 6)	N/A	N/A	

To install Raspberry Pi Zero to the interface, use the photo in illustration 2 as a reference.

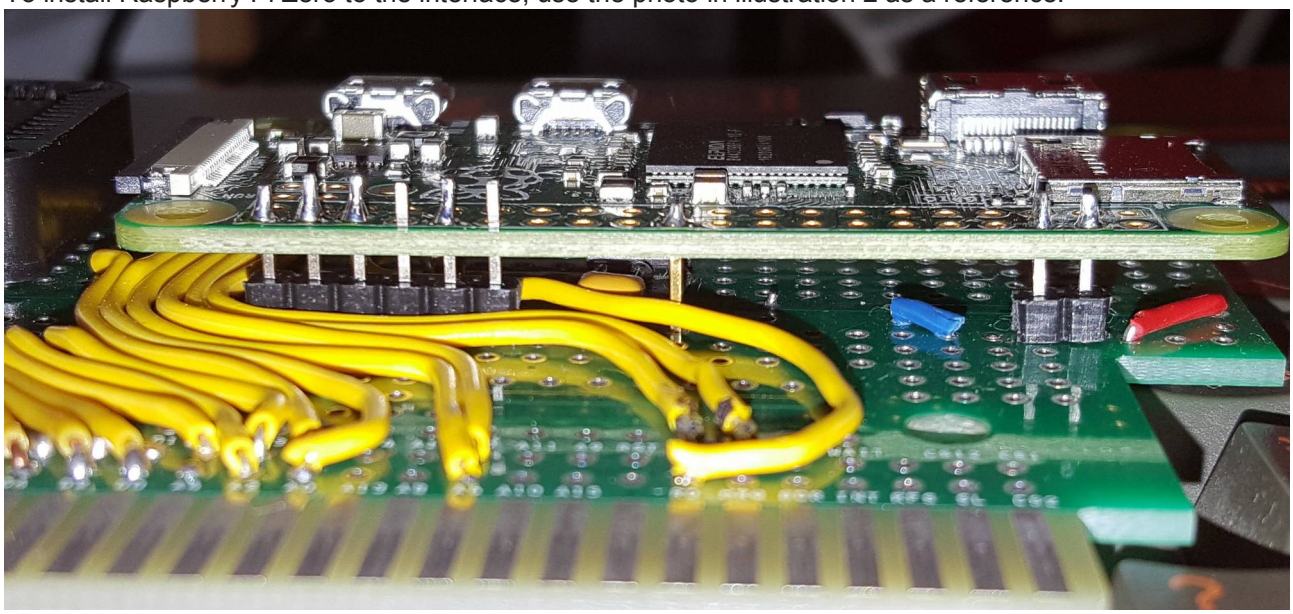


Figure 2: Installing Raspberry Pi Zero to the interface

To attach a Raspberry Pi model B +, 2 and 3 to the interface, use the picture in illustration 3 as a reference.

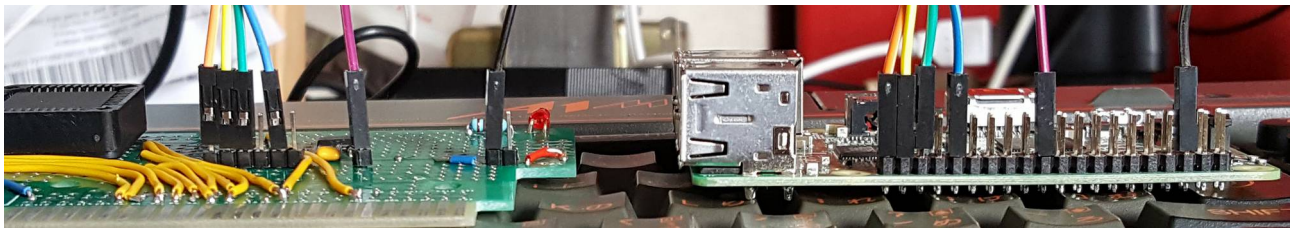


Illustration 3: Connection of the B +, 2 and 3 models

Note: It is essential that the ground wire (black wire) is connected from the Pi to the interface. Without this connection, the signals on the other interface pins will not be transmitted correctly.

4 MSXPi Programmer's Reference

This section describes the MSXPi specific IO library. This library is hardware-dependent and should be always be used instead of trying to access the hardware directly. This will allow for greater compatibility with future releases of the hardware.

The MSXPi software library consist of a set of Z80 assembly routines on the MSX side, and a C framework on the Raspberry Pi side. In this section we focus on the MSX-side library.

The MSX is broken down in the following set of files:

/asm-common/include/include.asm → contain BIOS and constant definitions

/asm-common/include/basic_stdio.asm → contain the PUTCHAR call for programs in BASIC environment and cartridge programs (such as MSX-DOS driver)

/asm-common/include/msxdos_stdio.asm → contain the PUTCHAR call for MSX-DOS programs, and also a MSX-DOS wrapper for the SENDPICMD bios routine.

/asm-common/include/msxpi_bios.asm → Contain all data transfer and other bios functions (hardware-independent) used by MSXpi programs.

/asm-common/include/msxpi_io.asm → This file contain the hardware-dependent I/O functions for MSXPi. All functions in this library access MSXPi hardware ports, and are called by other functions

/asm-common/include/msxpi_api.asm → contain extra-routines that might be useful.

Next section will describe the bios functions only.

4.1 MSXPi I/O (MSXPI_IO.ASM)

CHKSPIRDY

Label: CHKSPIRDY
Purpose : Read the MSXPi Interface status register (port 0x06)to determine the
status of the Interface and Pi Server App. This function should
return zero when Pi is responding, and 1 when not listening for commands.
Input : None
Output : Flag C: Set when Pi not responding.
Registers : AF is modified

Usage notes: Pi is determined as responding when I/O control port 06h returns zero. This routine loop 65535 times before it will return with error (zero not detected on that port).

PIEXCHANGEBYTE
Label: PIEXCHANGEBYTE
Purpose : Send a byte to Pi (write to data port 07H) and read a byte back. Usually,
the byte read at this cycle is meaningless because Pi send whatever is available

on its internal register during any transfer. If you send a byte expecting to receive a valid response, than you must call this function a second time (this is because Pi needs time to process your byte and execute whatever command it need to produce the expected answer).
Input : Register A: byte to send.
Output : Flag C: Set if there was an error
Register A: byte received
Registers : AF is modified

PIREADBYTE
Label: PIREADBYTE
Purpose : Read a byte from the MSXInterface. This command send value 0 to control port 06h, then call CHKSPIRDY to know when a byte is available to be read. In the sequence, the routine read data port 07h and return the byte in register A.
Input : None.
Output : Flag C: Set if there was an error
Register A: byte received
Registers : AF is modified

PIWRITEBYTE
Label: PIWRITEBYTE
Purpose : Send a byte to Pi. This function call CHKSPIRDY to know when Pi is available to receive data, and in the sequence write the data to the data port 07h.
Input : Register A: byte to send to Pi
Output : None
Registers : No registers are modified

SENDIFCMD
Label: SENDIFCMD
Purpose : Send a single-byte command to the MSXInterface (port 06h).
Input : Register A: Byte/Command to send.
Output : None.
Registers : None.

Usage notes: Current implemented commands are:

- Reset (0FFH) – Sending 0FF will force a RESET of the MSXPi interface internal state.
- Status (00h) – Reading this port will report the MSXPi state, being 0 = available.

4.2 MSXPi BIOS (MSXPI_BIOS.ASM)

SYNCH
Label: SYNCH
Purpose: Enforce a reset of MSXPi interface, and try to communicate with Pi.
Input: None

Output: Flag C: Set if connection failed.
Register A: Value returned by Pi if was able to connect, or error code.

Usage note: This function sends command “SYN” to Pi. It does nothing on Pi, but function a successful execution assure Pi is responding and waiting a command.

SENDPICMD
Label: SENDPICMD
Purpose: Send a command to Raspberry Pi
Input:
DE = should contain the command string
BC = number of bytes in the command string
Output:
Flag C set if there was a communication error
Modifies: AF, BC, DE, HL

RECVDATABLOCK
Label: RECVDATABLOCK
Purpose: Receive a block of data from PI. Calculate CRC using simple XOR of bytes received, and exchange with Pi at the end of the transfer.
Input:
DE = memory address to write the received data
Output:
Flag C set if error
A = error code
DE = Original address if routine finished in error,
DE = Next current address to read if finished successfully
Modifies: AF, BC, DE, HL

SENDDATABLOCK
Label: SENDDATABLOCK
Purpose: Send a number of bytes to Pi. Calculate CRC using simple XOR of bytes received, and exchange with Pi at the end of the transfer.
Input:
BC = number of bytes to send
DE = memory to start reading data
Output:
Flag C set if error
A = error code
DE = Original address if routine finished in error,
DE = Next current address to read if finished successfully
Modifies: AF, BC, DE, HL

SECRECVDATA
Label: SECRECVDATA
Purpose: Read data from Pi 512 bytes at a time.
Input:

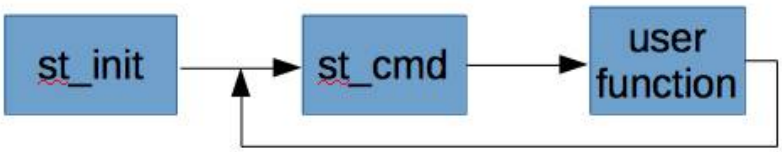
DE = memory address to start storing data
Output:
Flag C set if error
Modifies: AF, BC, DE, HL

SECSENDDATA
Label: SECSENDDATA
Purpose: Send data to Pi 512 bytes at a time.
Input:
BC = number of bytes to send
DE = memory to start reading data
Output:
Flag C set if error

DOWNLOADDATA
Label: DOWNLOADDATA
Purpose: Load data using configurable block size. Every call will read next block until data ends.
Input:
A = 1 to show dots for every 256 bytes
BC = block size to transfer
DE = Buffer to store data
Output:
Flag C: Set if occurred and error during transfer, such as CRC
Z: Set if end of data
Unset if there is still data
A: Error code
A = error code, or
A = RC_SUCCESS - block transferred, there is more data
A = ENDTRANSFER - end of transfer, no more data.
Modifies: AF, BC, DE, HL

4.3 Raspberry Pi

The server application that runs on Raspberry Pi is implemented using a state machine and functions implementing the commands, as shown in the following diagram.



The change of states occurs as commands are received by Pi. Each command should be parsed inside the “st_cmd” state, and call a function that implements the actions required by the command.

For the benefit of productivity, the re-implementation of the full server application is discouraged. The use of languages interpreted for performance reasons is also discouraged.

The low-level functions that communicate with MSXPi Interface are implementing using a bit-bang SPI-like protocol, thus it is serial. Due to this, performance is not impressive, but actually could be better. But it is an easy implementation using simple and cheap hardware.

The main state of the application is "st_cmd", which validates incoming commands, initializes all attributes necessary to execute each command, and call the specific function for the command.

The code structure that decodes a command and prepares the call to the function is shown below.

```
 } else if((strncmp(msxcommand,"PDIR",4)==0) ||
            (strncmp(msxcommand,"pdir",4)==0)) {
                printf("PDIR\n");
                if (pdir(msxcommand)!=RC_SUCCESS)
                    printf("!!!!! Error !!!!!\n");
                appstate = st_cmd;
                break;
```

For the given example above, the functions is implemented as show next:

```
int pdir(unsigned char * msxcommand) {
   memcpy(msxcommand,"ls  ",4);
   return runpicmd(msxcommand);
}
```

5 SPI Implementation in the MSXPi Interface

Communication between MSX and Raspberry Pi is facilitated by the MSXPi Interface, which implements a serial protocol using CPLD technology. The protocol uses five GPIO pins and full duplex transfers.
The GPIO pins implements the signals:

- CS (Enable signal from MSX, tells Pi that transfer should start)
- SCLK (Clock signal from Pi, tells MSX when to drive GPIO signals)
- MOSI (Data from MSX to Pi)
- MISO (Data from Pi to MSX)
- SPI_RDY (Ready signal, tells MSX when Pi is read to start a byte transfer)

When Pi is ready to process a transfer, it drive the SPI_RDY signal high.
When MSX wants to start a byte transfer, it checks SPI_RDY. If it is high, MSX drive CS low.
When CS toggle states, Pi jumps to an interrupt function to process the transfer if the signal went low. In this case, it also drive SPI_RDY low to tell MSX that it cannot send another byte.
PI starting generating the clock signal SCLK to synchronize the transfer, and send the 8 bits to MSX. MSX will receive each bit and use bit shift to store in a buffer.
When 8 bits are transferred, PI bring SPI_RDY high again to tell MSX that it can receive more data. The MSXInterface will keep the byte in its buffer. MSX can now send a read command to port 07h to get the byte.

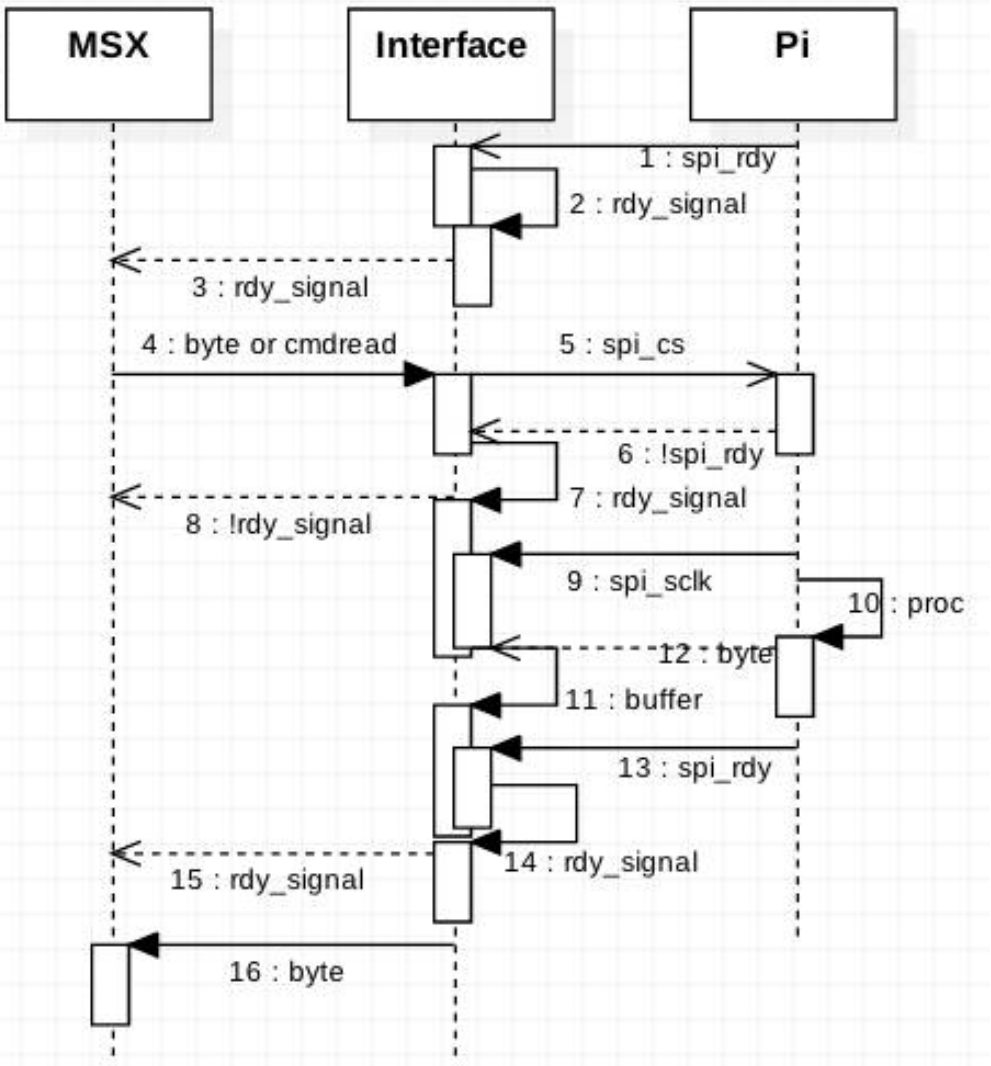


Illustration 2: Diagram Sequence for the Interface

6 Appendix 1: Development Tutorial

This appendix will be expanded (hopefully some day) with some useful contents.
As for now, please refer to the Documents/DevTemplate directory in the git repository for a template to use for development.
Also refer to the source codes available, specially under Client directory, for examples of implementation.

7 Appendix 2: Pi - GPIO Pin Numbering

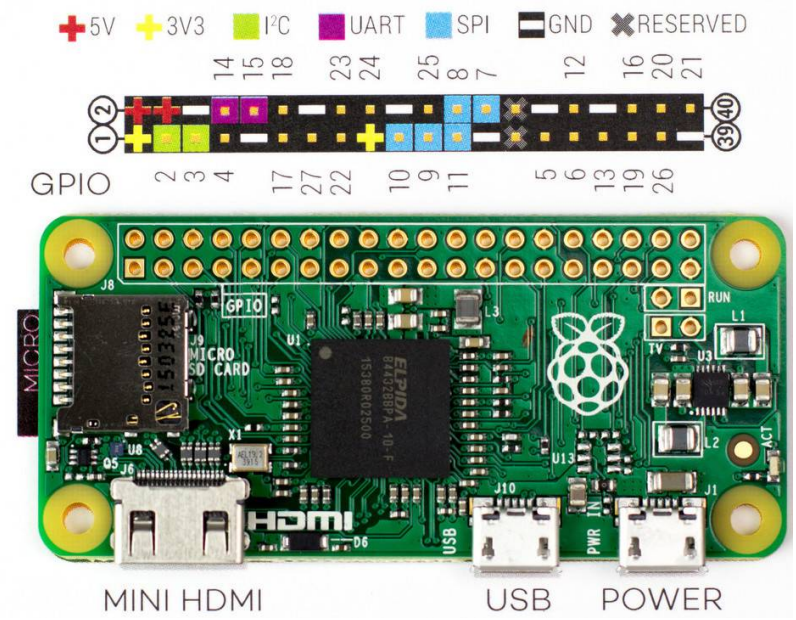


Illustration 4: GPIO models B +, 2, 3 and Zero.

The GPIO numbering of illustration 4 is valid for B +, 2, 3 and Zero models.

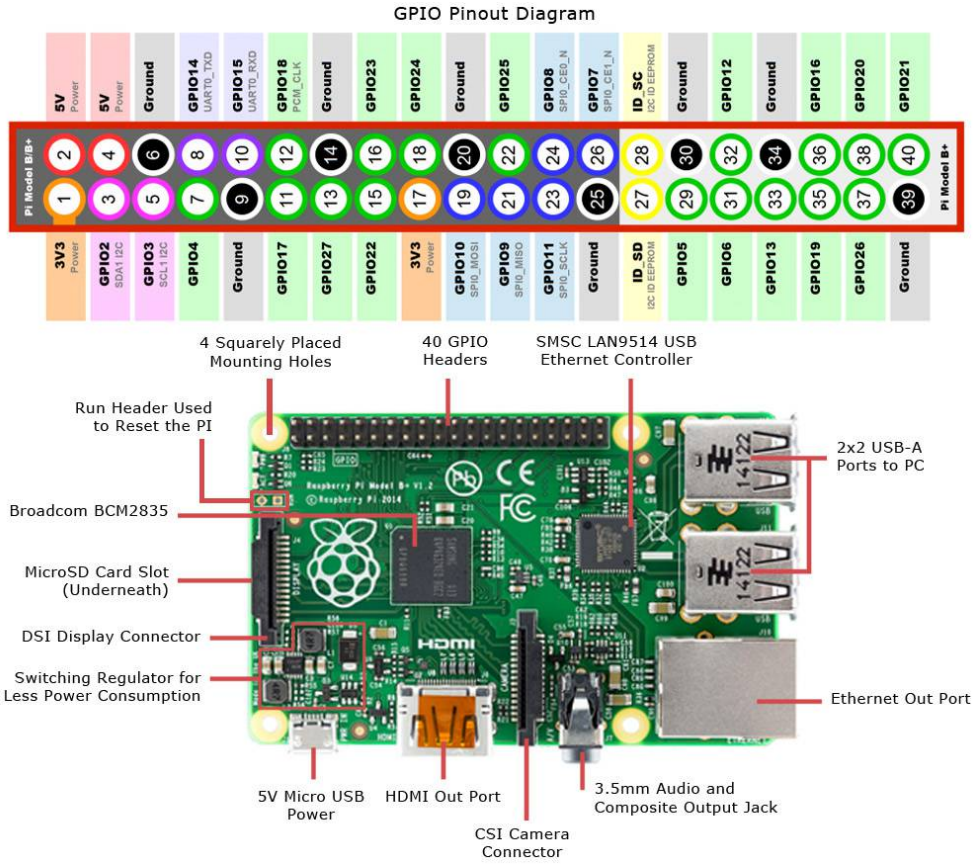


Illustration 5: GPIO models B +, 2, 3 and Zero.