MSX Pi Interface v. 0.7
Ronivon Candido Costa
ronivon@outlook.com

Technical Reference Manual

Table of Contents

1Users Guide.......... 3
1.1 For ROM-less MSXPi.......... 3
1.2 MSXPi Client Commands.......... 3
2Architecture and Behaviour Description.......... 7
3Connecting MSXPi to the Raspberry Pi.......... 9
4MSXPi Programmer's Reference.......... 11
4.1MSXPi BIOS.......... 11
4.2MSXPi API.......... 12
4.3MSXPi Client.......... 18
4.4Raspberry Pi.......... 18
5SPI Implementation in the MSXPi Interface.......... 20
6Appendix 1: Development Tutorial.......... 22
6.1Implementation Details on MSX.......... 22
6.2Implementation Details on Raspberry Pi.......... 23
7Appendix 2: Pi - GPIO Pin Numbering.......... 26

NOTE: I apologize for the low quality of this translation. I rushed to get it available in English for our international contributors, and had to use google translation for the job. I am committed to improve this translation by doing it manually asap.
Section 3 was translated manually, so any bad translation on that section should be blamed on me, not google.

1 Users Guide

```
From the BASIC prompt, type any of these commands:

CALL HELP
CALL MEMMAP
CALL MSXPIHELP
CALL MSXPISTATUS
CALL MSXPILOAD
```

1.1 For ROM-less MSXPi

The very first prototypes of the MSXPi did not have a ROM or BIOS. For this reason, instead of calling “call msxpiload” to load the client, it is needed to type in a small BASIC program to load the the client software.
Before running the loader, it is advised to check if Pi status:

```
print inp (6)
```

If a response to "0", Pi is available to receive commands.
If for "1", it is not available (it may still be still booting, or the MSXPi Server app did not start up for any reason).

When you receive "0" as a response, ROM loader can be started:

```
out 7,1
for i = 0 to 205:out 6,0: poke &hC000+ i, inp(7):next: defusr=&hC000: a=usr(0)
```

The rom will be loaded and started.

1.2 MSXPi Client Commands

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

```
MSXPi Interface v0.7
MSXPi Client v0.7
TYPE HELP for available commands
Raspberry PI is online
CMD:
```

The available commands can be viewed with the HELP command:

CMD:HELP
BASIC CHKPICONN CD"<url>|dir" CLS DIR HELP LOAD"<url or file>""<,R> MORE #(Pi Command) PIAPPSTATE #(Pi command) PIPOWEROFF PWD RESET SET WIFI

CD: will change the current path. Valid values are filesystem path, or remote path. Currently remote paths suported are:

http://
Example: cd http://www.msxarchive.nl:80/pub/msx

Example: cd ftp://ftp.funet.fi/pub

Example: cd nfs://10.1.1.50/shared_filesystem

Example: cd win://muywindowsxp.local/Users/myuser/Public

And of course local filesystems are supported.
Example: cd /usr/local/games

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 filesyste, and also remote

LOAD: Load a binary program into memory. Curreently supported formats are binary files to run from BASIC, and .rom (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 works for both full path and relative paths – in this case, file will be loaded from current path set with CD command. It loads files from Pi filesystem or remote systems supported by CD command.

Examples of usage:
LOAD"EXERION.BIN",R
LOAD"http://192.168.1.14:8000/EXERION.BIN",R
cd win://10.10.10.1/Users/roni/Public/msx
LOAD "GALAGA.ROM",r

Note: If ",R" is not used, file is loaded into correct memory address and returns to CMD: prompt.

#: 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

PIPOWEROFF: Shutdown the Raspberry Pi. This command should also be called when running ROM games, to shutdown the Pi before running the game. This should be done because it is not possible to exit ROM games, being necessary to switch off the MSX. For the MSXPi with integrated Pi Zero, when the MSX is switched off, the MSXPi is also switched off since its mains power comes from the MSX Slot. This could eventually damage the filesystem in the SD Card.

PWD: Display current path or url.

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

SET: Modify or display user parameters in the MSXPi Server. These attributes can be viewed with command "set display".
Current supported parameters are:

- msx_path1, which contains the current path set with command CD
- remoteuser, which contains a username used to authenticate into remote systems
- remotepass, which contain the password for the remoteuser above.
- wifissid, which contain current set wifi network id
- wifipass, which contain current set wifi password

WIFI: Configure wifi network on Pi, or display current configuration. Before using this command, the wifissid and wifipass must be set using the "SET" command. There are three parameters that can be used:

- "display" to show current network interface configuration
- "add" (to append a new wifi network to existing configuration)
- "replace" to rewrite wifi configuration. Using "replace" will erase all existing wifi configuration with a single wifi configuration.

Examples of use:

To replace all wifi configuration with a new one:
set wifissid=roniwifi
set wifipass=secret
wifi replace wlan0

To append a new wifi network to existing wifi configuration:
set wifissid=friendwifi
set wifipass=friendsecret
wifi add wlan0

wifi display

Note: Currently only wlan0 and wlan1 are supported. This parameter in the “wifi” command is used only to restart the interface after configuration is applied to /etc/wpa_supplicant/wpa_supplicant.conf.

2 Architecture and Behaviour Description

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

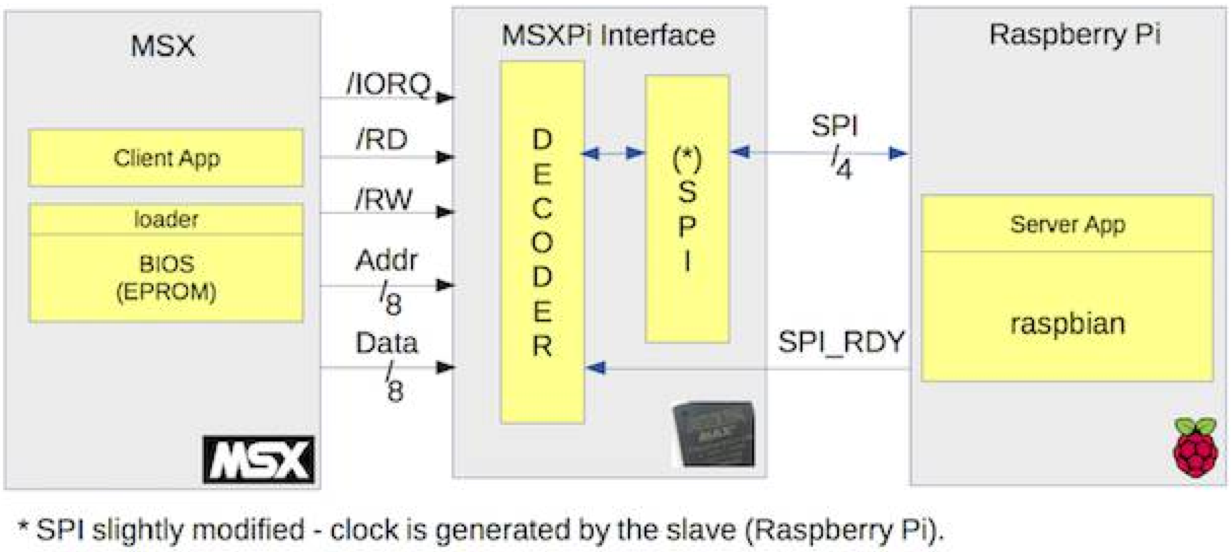


Ilustração 1: Arquitetura

1. MSX starts a transfer with Pi, by one of two means:
1a. MSX send a read signal to Pi, by writting 0 to port 0x06;
1b. MSX send a write signal to Pi, by writing a value to port 0x07.
2. CPLD decodes address 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 0x07 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).

As an illustration, the interface in the current version does not have a built-in ROM, and for that reason, the first step in using it is to use the charger0 to read a second charger1, which in turn will read the final rom and run. The loader0 is a BASIC line, which needs to be typed (or loaded from another mass storage device such as SD / Flash drive). Loader1 is a small code in assembly, to be loaded quickly by loader0 which is very slow because it is in BASIC. The charger1 has the sole purpose of loading the final ROM and executes it, which is very fast because it is in assembly, even when the ROM is too large.
A demonstration of this process can be visualized in the video published in:

https://www.youtube.com/watch?v=SkwLT3p6uwg

Communication between the MSX and the Pi occurs through a protocol. This protocol will be described later in this paper, but it is important to mention now that data transfers between MSX and Pi can occur in "one cycle" or "two cycles".

- A Cycle: Occurs during data transfer (that is, the byte is not a command), such as transferring a file from Pi to MSX. A one-cycle process must pre-establish a sub-protocol such as setting a number of bytes to be transferred before initiating the transfer. In this way, bytes can be sent sequentially since both parties know when the transfer should finish.
- Two Cycles: It is any byte transfer that occurs when the server application is in command state. The bytes received by Pi are interpreted as commands that require an acknowledgment so MSX knows that it can send more commands or data.

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, via an entry in /etc/rc.local.

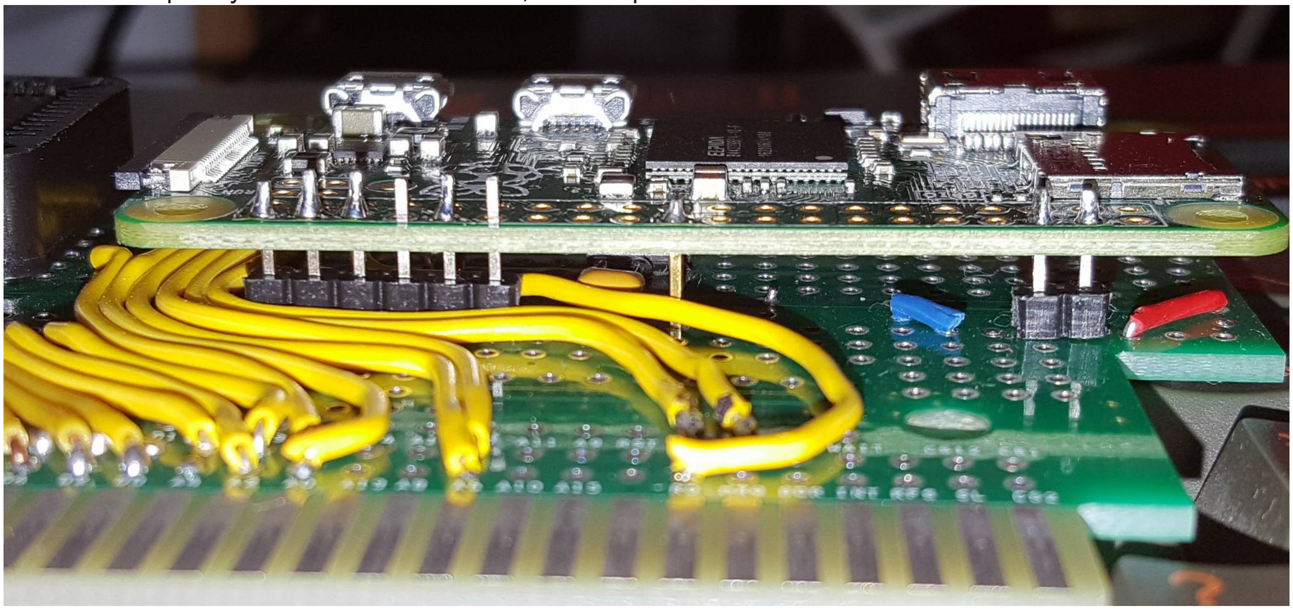
By using a standard linux system, the solution allows a wide range of applications to be easily developed through the use of existing tools in Raspberry, both for programming and for accessing Pi GPIOs.

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.



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

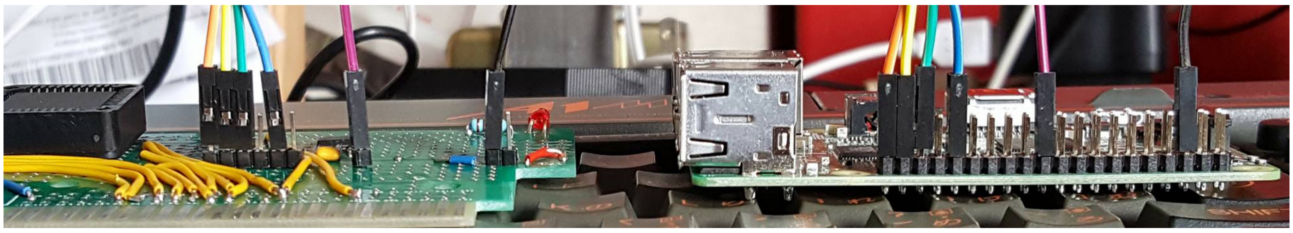


Illustration 3: Connection of the B +, 2 and 3 models

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

4.1 MSXPi BIOS

CHKSPIRDY

Label: CHKSPIRDY

Purpose	: Read the MSXPi Interface status register (port 0x06)to determine the status of the Interfeface 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 or other error occurred.
Registers	: AF

READBYTE

Label: READBYTE

Purpose	: Read a byte from the MSXInterface. This command send value 0 to port 0x06, then call CHKSPIRDY to know when the byte is available. In the sequence, the routine reasd port 0x07 and returnt the byte in register A.
Input	: None.
Output	: Flag C: Set if there was an error. Register A: byte received.
Registers	: AF

TRANSFBYTE

Label: TRANSFBYTE

Purpose	: Send a byte to Pi (write to port 0x07) and read a byte back. Usually, the byte read at this cycle is meaningless, unless the program has set a protocol to receive data delayed by one cycle (remember that data is serialized inside the CPLD, sent to Pi, which must process return the requested data to the MSX. This needs at least two protocol cycles).
Input	: Register A: byte to send.
Output	: Flag C: Set if there was an error. Register A: byte received.
Registers	: AF

SENDIFCMD

Label: SENDIFCMD
Purpose : Send a single-byte command to the MSXInterface (port 0x06).
Input : Register A: Command to send. Current implemented command is 0xFF (Reset).
Output : None.
Registers : None.

4.2 MSXPi API

LOAD

Label: LOAD
Purpose : Load a file from Pi into MSX memory. The command transparently receives the file from any supported location, as implemented in the server: Pi local file, http, ftp, nfs, windows share.
File name can be relative to current path (set using CD command) or a complete url or path
Optionally, there can be a ",R" after the filename. This will make the MSXPi client to try to execute the program after loading.
Currently, .bin files and some .rom files are supported.
Input : Register HL: Memory address to store the file (only when RAW file is being loaded).If file type is .bin or .rom, the exec address is extracted from the file. For .rom files, they are stored in the target address using inter-slots write operations, which delays the loading process.
Register DE:Memory address with the filename or url to load.
Output : Flag C: Set if there was an error.
Register A:File type as detected by the routine.
Register HL: Exec address as extracted from the file header.
(AUTORUN) will contain value 1 if file name had ",R"
Registers : AF,BC,DE,HL
File types : 0 = RAW (no header in file).
1 = Reserved (id for the MSXPi Loader).
2 = Reserved (id of the MSXPi Client)
3 = Basic binary fil (.bin, where header is
#FE, StartAddr,EndAddr,ExecAddr)
4 = Cartridge file (.rom file with headers starting in #41,#42)

Usage : Consult LOAD command in the MSXPi Client sources.

PARSECMD

Label: PARSECMD
Purpose : Check if string pointed by DE is a valid command. Verification is done against a list of commands starting in (COMMANDLIST).
Input : Register HL: Command (string) to check.
(COMMANDLIST): This memory address contain the list of valid commands. Commands are strings, ending in zero, followed by the exec address (two bytes). This pattern repeats. The list terminates when there is a zero where it should have been a new command.
Output : Flag C: Set when command is invalid or not found
Register A: Set to value 1 when command is valid, and also to flag that there could be a parameter after the command.
Register DE: Memory addres of a (potencial) parameter in the command.
Register HL: Exec address of the command. After returning from this function, the program can jump to the HL address to run the command:
jp (hl).
Registers : AF,BC,DE,HL
Usage : Read the “PROGLOOP” section the the MSXPi Client source code.

PARSEPARM

Label: PARSEPARM
Purpose : Gets a memory address in DE and verify the the string starting at that address is a valid parameter. Parameters should follow after a command, and can be delimited by space or quotes. Parameters terminates when zero in found. Parameters can include a “,R” (without the quotes) when loading a program, so the program will be executed.
Input : Register DE: Memory address where the parameter is stored.
Output : Flag C - set if there is not parameter, or is invalid
A = 0 there is not any parameter
A = 1 there is a valid parameter
A = 2 there is a potentially valid parameter (terminate without quotes)
A = 255 syntax error in parameter

(PARMBUF): This memory address contain the verified parameter address.
(AUTORUN): this memory address contain 1 if “,R” was found, 0 otherwise.
Registers : AF,BC,DE,(AUTORUN),(PARMBUF)
Usage : Consult the LOAD command in the MSXPi Client.

PISERVERSHUT

Label : PISERVERSHUT
Purpose : Send shutdown command to Pi (0x66).
Input : None.
Output : Flag C: Set if there was an error.

Registers	: AF,BC,DE

PRINT

Label	: PRINT
Purpose	: Print a string in the screen. Continue printing from last known cursor position.
Input	: Register HL: Address of string to print, terminated by zero.
Output	: String is printed on screen.
Registers	: AF,HL.

PRINTNLINE

Label	: PRINTNLINE
Purpose	: Send a new line to screen.
Input	: None.
Output	: Cursor goes to start of next line.
Registers	: AF

PRINTNUMBER

Label	: PRINTNUMBER
Purpose	: Print a hexadecimal value (0 to 0XFF) in the screen.
Input	: Register A: Value to print.
Output	: None.
Registers	: AF

PRINTDIGIT

Label	: PRINTDIGIT
Purpose	: Print a single-digit hexadecimal value (0 to 0xF) in the screen.
Input	: Register A: Value to print.
Output	: None.
Registers	: AF

PIAPPRESET

Label	: PIAPPRESET
Purpose	: Send reset command to Pi, which consists of command 0x66 and a sequence of characters to help recover from unexpected status. This includes send quotes to

inform Pi that it should stop waiting for parameters, for example. This sequence of bytes will put the CPLD in ready state, and try to enforce the MSXPi server app to go to st_cmd state, where it will be listening for new commands.

```
Input       : None.
Output      : None.
Registers   : AF,BC
```

PG0RAMSEARCH

```
Label       : PG0RAMSEARCH
Purpose     : Search for slot/sub-slot where RAM page 0 (0x0000) is allocated. Should work for any
              MSX model, and also for expanded slots as well. The output is in register A, where it
              can be used directly to call inter-slots rutines such as RDSLT, WRSLT and others.
Input       : Register C = 0x00
Output      : A  - ExxxSSPP
               |   ||Primary slot number  (00-11)
               |   - Secundary slot number (00-11)
              +----------- Expanded slot (0 = no, 1 = yes
Registers  : AF,C,DE
```

PG1RAMSEARCH

```
Label       : PG1RAMSEARCH
Purpose     : Search for slot/sub-slot where RAM page 1 (0x4000) is allocated. Should work for any
              MSX model, and also for expanded slots as well. The output is in register A, where it
              can be used directly to call inter-slots rutines such as RDSLT, WRSLT and others.
Input       : Register C = 0x40
Output      : A  - ExxxSSPP
               |   ||Primary slot number  (00-11)
               |   - Secundary slot number (00-11)
              +----------- Expanded slot (0 = no, 1 = yes
Registers  : AF,C,DE
```

PG2RAMSEARCH

```
Label       : PG2RAMSEARCH
Purpose     : Search for slot/sub-slot where RAM page 2 (0x8000) is allocated. Should work for any
              MSX model, and also for expanded slots as well. The output is in register A, where it
              can be used directly to call inter-slots rutines such as RDSLT, WRSLT and others.
Input       : None
Output      : A  - ExxxSSPP
               |   ||Primary slot number  (00-11)
               |   - Secundary slot number (00-11)
              +----------- Expanded slot (0 = no, 1 = yes
```

Registers : AF,C,DE

PG3RAMSEARCH

Label   : PG3RAMSEARCH
Purpose   : Search for slot/sub-slot where RAM page 3 (0xC000) is allocated. Should work for any MSX model, and also for expanded slots as well. The output is in register A, where it can be used directly to call inter-slots rutines such as RDSLT, WRSLT and others.
Input   : None
Output   : A - ExxxSSPP

```
  |   ||Primary slot number  (00-11)
  |   - Secundary slot number (00-11)
  +----------- Expanded slot (0 = no, 1 = yes
```

Registers : AF,C,DE

READCMD

Label   : READCMD
Purpose   : “Read Command”, will call INLIN function to read a string from keyboard. The prompt “CMD.” is shown on screen – to avoid printing the prompt, call “READCMD0”.
Input   : None.
Output   : Register HL: Address of buffer with the string.
Registers : AF,BC

READTEXTSTREAM

Label   : READTEXTSTREAM
Purpose   : Read a stream of ascii characters from Pi. The stream should finish with zero.
This function must not be used to read binary data.
The ascii code 10 is converted into ascii codes 10 and 13 (start of new line).
Input   : None.
Output   : The data received is printed to the screen as characters (text).
Registers : AF,BC,HL
Usage   : See command “DIR” in the MSXPi client.

RECVBINDATA

Label: RECVBINDATA
Purpose   : Read a sequence of bytes from Pi and stores in memory.
Input   : Register DE: Initial memory address to start writing data.
Register BC: Number of bytes to read.
FLAG C set to progression dots every 256 bytes.

Output : Memory starting in DE will receive BC bytes read from Pi.
Flag C: Sey if there was an error.
Registers : AF,BC,HL,DE

RECVTXTDATA

Label: RECVTXTDATA
Purpose : Read a sequence of ASCII bytes from Pi and stores in memory.
Input : Register DE: Address to buffer to save data.
Register BC: Number of bytes to read.
FLAG C set to progression dots every 256 bytes.
Output : Memory starting in DE will receive BC bytes read from Pi.
A = 0 if there is not more data to read (received zero from Pi)
Flag C: Sey if there was an error.
Registers : AF,BC,HL,DE

SENDPARMS

Label : SENDPARMS
Purpose : Send a stream of data to Pi. Read data in memory address pointed by HL until finds zero or quote.
Input : (PARMBUF) memory address should contain the address of the buffer to start reading: ld hl,(PARMBUF).
Register HL: As a second option, the buffer can be set directly in HL, then calling SENDPARM0.
Registers : AF,BC,HL

SETPARM

Label : SETPARM
Purpose : Send parameter to Raspberry Pi.
Pi Server app has got a internal variable “buf_parm” that is used in several commands. This function will set this variable (in memory only). Commands such as LOAD and DIR make use of this variable. No verification is done to the parameters, therefore you should call PARSEPARM before calling SETPARM.
Input : (PARMBUF): Contain the memory address where the buffer starts.
Output : Flag C: Set if there was an error.
Registers : AF,BC,HL

SENDPICMD

Label : SENDPICMD
Purpose : Send a single-byte command to PI, and waits acknowledge.

A delay is inserted in the waiting loop, to give Pi to process the command.
Acceptable responses from Pi ACK, PROCESSING and CMDERROR.
Everything else is considered as an error.

Input	: Register A: Command to send (there is a list defined in the MSXPi client).
Output	: Flag C: Set if there was a communication error. Register A: ACK if command executed successfully (ACK is the same byte sent). PROCESSING (0xAE) if Pi is still busy processing the command. CMDERROR (0xEE) if command was executed but resulted in error.
Registers	: AF,BC,DE

SHOWDOTS

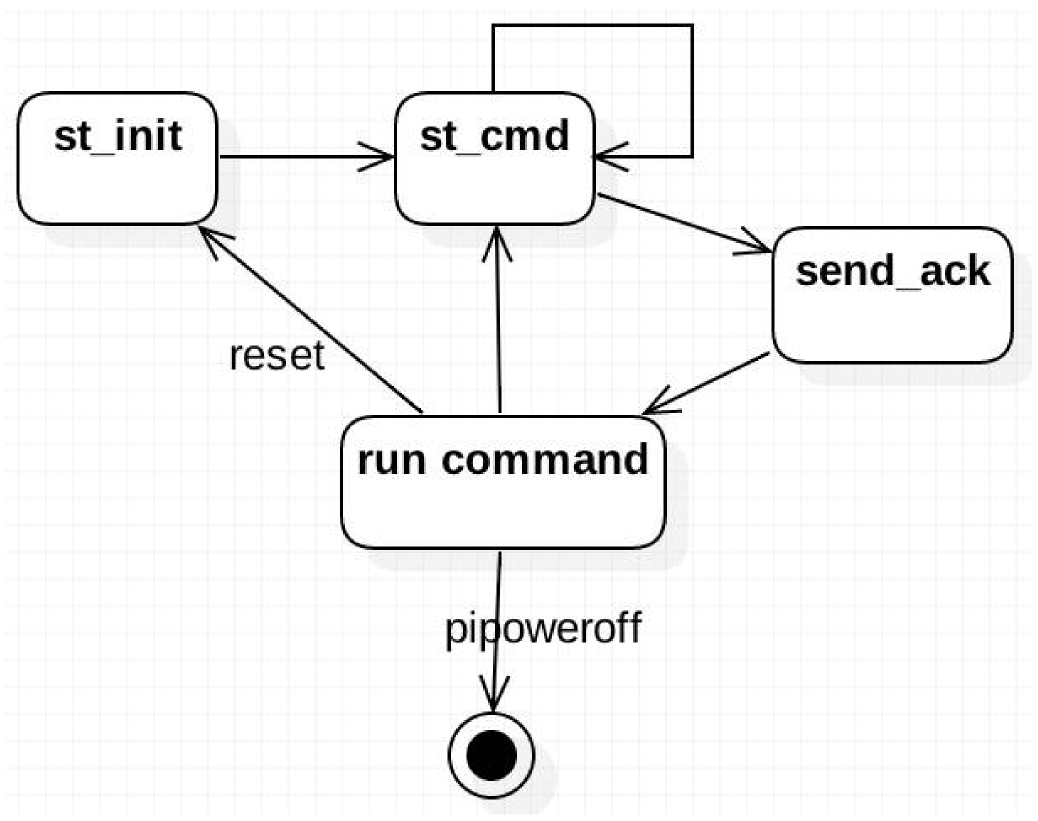
Label	: SHOWDOTS
Purpose	: Display download progression on screen, one dot for every 255 bytes. Memory address (CNT1) is used to store the temporary values during download.
Input	: None.
Output	: None.
Registers	: AF, (CNT1)

4.3 MSXPi Client

To-do.

4.4 Raspberry Pi

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



The application has more states than shown in the diagram. Each implemented command has its own state, and can also make use of other intermediate states and functions.
The change of states occurs as commands are received by Pi, it has a high level of complexity as it needs to follow a logical sequencing, chained and taking into account the previous state.
The recommended way to implement new commands is to append the new states to the new commands, using existing commands as templates.
For the benefit of productivity, the re-implementation of the server application is discouraged. The use of languages interpreted for performance reasons is also discouraged.

The main state of the application is "st_cmd", which validates incoming commands, initializes all attributes necessary to execute each command, and changes the state of the state machine so that the command is triggered in the next cycle.

The code structure that decodes a command and prepares the state transition is shown below, although not all commands need to follow this rule.

```
1    case CMDCHKPICONN:
2         printf("Received command CHECKPICONN 0x%x\n",msxbyte);
3         fprintf(flog,"Received command CHECKPICONN 0x%x\n",msxbyte);
4         msxbyterdy = 0;
5         pibyte = msxbyte;
6         appstate = st_send_ack;
7         nextstate = st_cmd;
8         nextbyte = appstate;
9         gpioWrite(rdy,HIGH);
10        break;
```

Line 4: The "msxbyterdy" flag is set to "0" so that no other state is executed, and switched to "1" only by Purpose of reading data every time a byte transfer occurs. With msxbyterdy "at" 0 ", keeping the" rdy "signal at" LOW "inhibits the transfer of a new byte, in order to allow MSXPi

Server time to complete the current processing. It is for this reason that at the end of each state there is a "gpioWrite (rdy, HIGH)", because once the current state processing is finished, the transfer of new data or commands can be released.
Line 5: "pibyte" is loaded with the value of the command that was received. Under the established protocol, the MSX sends a command, and expects to receive the same command in the response. This is the "ack" defined in the protocol.
Line 6: "appstate" is the next state of the state machine. According to the protocol, it is necessary to send an "ack" to the MSX after a command, so "appstate" is loaded with the state "st_send_ack", which will be the status machine state in the next cycle.
Line 7: "nextstate" is the next state after "st_send_ack". This variable is required because st_send_ack needs to know what to do (for what state to go) after sending the acknowledge to MSX. Thus, "appstate" tells the state of the next cycle, and "nextstate" tells the state of the cycle after the next cycle.
Line 8: "nextbyte" has Purpose parallel to "nextstate", but for the byte to be sent. In other words, "pibyte" contains the next byte to be sent, and "nextbyte" contains the byte to be sent in the second cycle after the current cycle. This attribute is not required in many states.
Line 9: The "SPI_RDY" signal is set to "1", to indicate to Interface that Pi is ready to process another transfer. If this command is not present, the signal will remain at "0" indefinitely, indicating that Pi is unavailable, and therefore no data transfer is possible.

5 SPI Implementation in the MSXPi Interface

Uma comunicação entre uma interface e uma framboesa via protocolo SPI e mais um sinal de controle SPI_RDY.
Ao receber um evento "falling_edge" não pino SPI_CS, o Pi aciona uma interrupção para um propósito "func_st_cmd", que por sua vez baixa o sinal não pino SPI_RDY para que uma interface que não pode receber outros comandos. Em seguida, o byte a ser transferido "pibyte" usado como parâmetro na chamada da Finalidade "SPI_MASTER_transfer_byte" que efetivamente serializa e transfere o byte para a Interface via sinal SPI_MISO, mesmo tempo que recebe o byte da Interface / MSX via sinal SPI_MOSI .
Tanto o byte enviado quanto o byte recebido são armazenados em variáveis globais, acessíveis dentro da Finalidade e também não programa principal dentro da máquina de estados.

É importante não alterar uma estrutura da máquina de estados, bem como o sequenciamento de estados via "appstate", "nextstate", "msxbyterdy", pois como interrupções podem acontecer a qualquer instante, causando instabilidade na aplicação. Da mesma forma, "gpioWrite (rdy, HIGH)" só pode ser usado quando MSX para enviar novos comandos ao Pi. "GpioWrite (rdy, LOW)" pode também ser usado em partes específicas do código em uma medida necessária para inibir o envio de novos comandos ou dados durante o processo.
O diagrama de sequência da implementação da interface para leituras e escritas de dados está ilustrado no diagrama 2 abaixo.

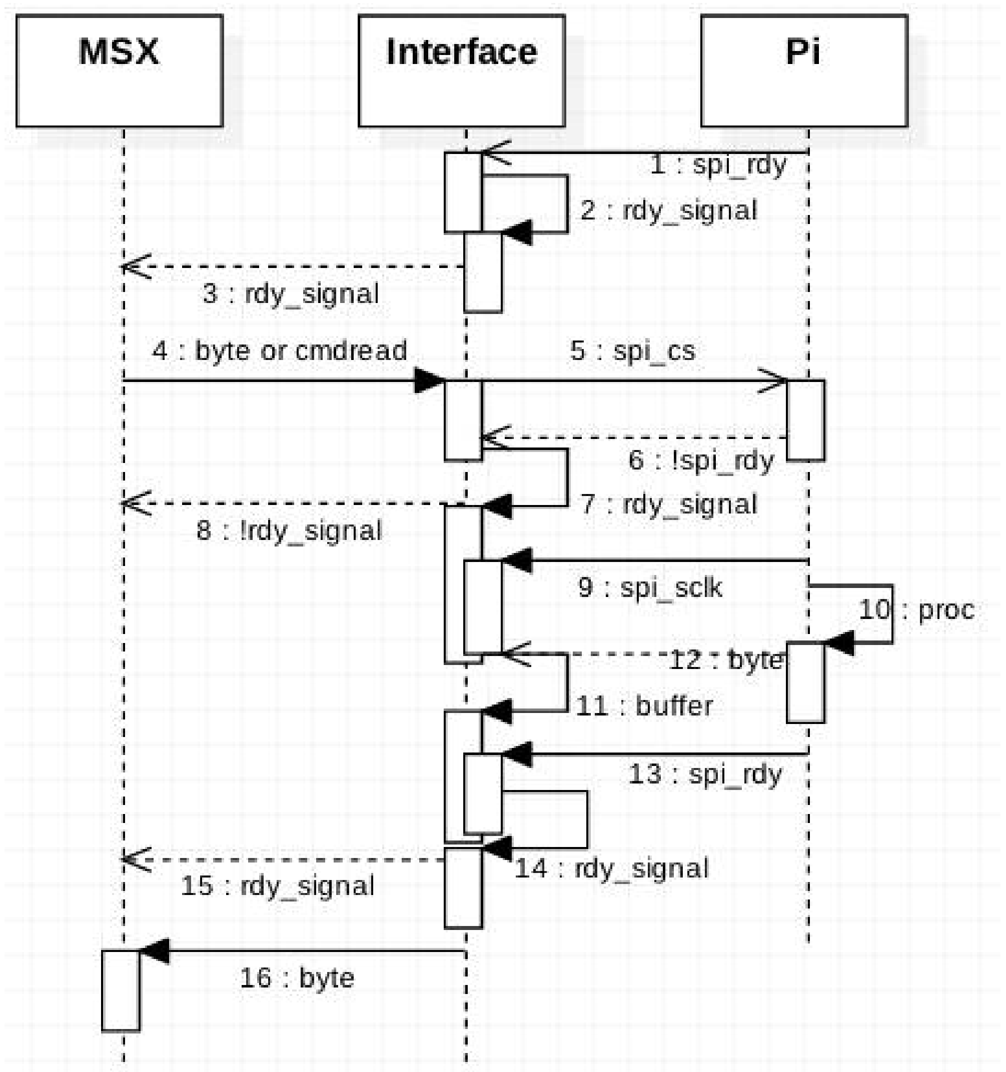


Illustration 2: Diagram Sequence for the Interface

6 Appendix 1: Development Tutorial

This appendix will demonstrate how to use an interface software library to accelerate application development. Let's use as an example the implementation of command # (execution of remote commands in Pi).
The # command has the following characteristics:

- Requires the parameter (ie the command to send to Pi)
- Require a part to be run on Pi (the command)
- Receiving an indeterminate number of characters in response

These features are common to many applications that have been implemented, so they are a good example to follow.

6.1 Implementation Details on MSX

Let's start the tutorial by creating the MSX-side application.
First, it is necessary to add a new command to the MSXPi Client, which is done by adding these two lines to the code:

```
RUNPICMDSTR:
              DB      "#",0
              DW      RUNPICMD
```

Table A1: Defining New Commands

This is sufficient for the new command to be recognized by the client. But for it to actually work, wee must implement the code for the command. In our example, the command is implemented by this code:

```
RUNPICMD:
            CALL    PARSEPARM
            JP      C,PRINTPARMERR
            CALL    SETPARM
            JP      C,PRINTPARMERR
            LD      A,(CMDRUNPICMD)
            CALL    SENDPICMD
            JP      C,PRINTPIERR
            SCF
            CALL    READSTREAM0
            RET
```

Table A2: New Command Implementation

The label "RUNPICMD" in table A2 corresponds to the label of the command DW in table A1. DW indicates the execution address of the implemented command.

Our # command needs parameters (the command to execute in Pi). For example, "ls -l".

In line (2), the parameter is checked and if it is considered invalid (line 3), an error message is displayed on the screen, and the program returns to the "CMD:" prompt. We are here using the "PRINTPARMERR" subroutine to show the error message, the "RET" at the end of it causes our program to return to the input loop of new commands.

Line (4) sends the parameters found after the # to the Pi, which will in turn store in the variable "buf_parm". Subsequently (line 7), this parameter will be used.

Line (5) evaluates the return code of the command, and if it is an error, prints an error message and returns to the "CMD:" prompt, in the same way as described above for line (3).

Line (6) gets the command code "#".

Line (7) sends the command to Pi. SENDPICMD sends the command to register A and waits for Pi to execute. It then sends a read-only command, out (6), 0 to read the ACK sent by Pi.

Line (8) evaluates the execution status of the command. If it is an error, it prints an error message and returns to the "CMD:" prompt.

Line (9) sets C flag to inform READSTREAM to pause scroll.

Line (10) Reads the stream of characters from Pi, and prints on the screen, character by character as it receives, until it receives zero (0).

Line (11) Returns the command prompt "CMD:".

6.2 Implementation Details on Raspberry Pi

Each command receives a unique identifier, as seen above. And in some cases, a new state needs to be created in the "state machine" that implements the application logic in Pi.
In the case of the # command it has code 13, and an additional state has been created. The following two lines have been added to msx.c:

```
#define st_msxpiruncmd          13
#define RUNPICMD                0x22
```

The command # is then implemented using two states, the first being the detection and execution of the command, which occurs in the state "st_cmd", and the second state is the "st_file_send", responsible for sending the text with the result of the command back To MSX.

Within the "st_cmd" state, the case was added to the # command:

```
case RUNPICMD:
     printf("Received command RUNPICMD 0x%x\n",msxbyte);
```

```
 3      fprintf(flog,"Received command RUNPICMD 0x%x\n",msxbyte);
 4
 5     if(!(fp = popen(buf_parm, "r")))
 6         {
 7             printf("Command failed\n");
 8             fprintf(flog,"Command failed\n");
 9             pibyte = UNKERR;
10             msxbyterdy = 0;
11             appstate = st_send_ack;
12             nextstate = st_cmd;
13             gpioWrite(rdy,HIGH);
14             break;
15         }
16
17         fread(msx_pi_so, sizeof(char), sizeof(char) * sizeof(msx_pi_so), fp);
18
19         printf("Command executed successfully\n");
20         fprintf(flog,"Command executed successfully\n");
21
22         //printf("Command output:\n%s\n",msx_pi_so);
23
24         fileindex = 0;
25         filesize = strlen(msx_pi_so);
26
27         appstate = st_send_ack;
28         nextstate = st_file_send;
29         lastcmd = RUNPICMD;
30         TRANSFTIMEOUT = PIWAITTIMEOUTOTHER;
31         // prepare to check time
32         time(&start_t);
33
34         msxbyterdy = 0;
35         pibyte = msxbyte;
36         nextbyte = DATATRANSF;
37
38         gpioWrite(rdy,HIGH);
39         break;
```

The command sent by MSX is executed on line 5. Note that "buf_parm" was defined by "CALL SETPARM" of line (4) in Table A2.

In line (17), the text generated by the command (for example, in the case of an "ls -l") is stored in the char array variable "msx_pi_so". This variable is used by the application to store data that will be serially sent to Pi.

Line 24 starts the file indexer, since each byte will be read individually and sent to the MSX. This indexer starts at zero, the first byte of the file.

Line 25 calculates the size of the file, because the loop that controls the sending of the bytes will use this information to know when to stop sending data to the MSX, or when you have read all the bytes. The routine that will read, send, and verify when stop sending data is implemented in the "st_file_send" state. Refer to this routine in the code, and you will see that it ends when "fileindex + 1> filesize".

Reading each byte in "msx_pi_so" occurs in Purpose "func_st_cmd". If you refer back to section 3, you will see that this routine is triggered by interrupt, whenever MSX does an "out (6), 0" or "out (7), nn". Any of these "out" causes the CPLD to trigger the signal / pin "CS" in the Pi, generating the interrupt.

At each interrupt generated, one byte is read from "msx_pi_so", fileidex is incremented, and the loop continues until all characters (or binary bytes, in some cases) have been transferred.

NOTE: The rest of the code is standardized in all routines, but it is important to mention that it is line 38 that sends the "SPI_READY" signal to the CPLD, which in turn returns zero to the MSX when a command "IN A, 6) "is executed. It is this control that keeps MSX "waiting" for Pi to complete the processing it is executing before sending more commands or data.

7 Appendix 2: Pi - GPIO Pin Numbering

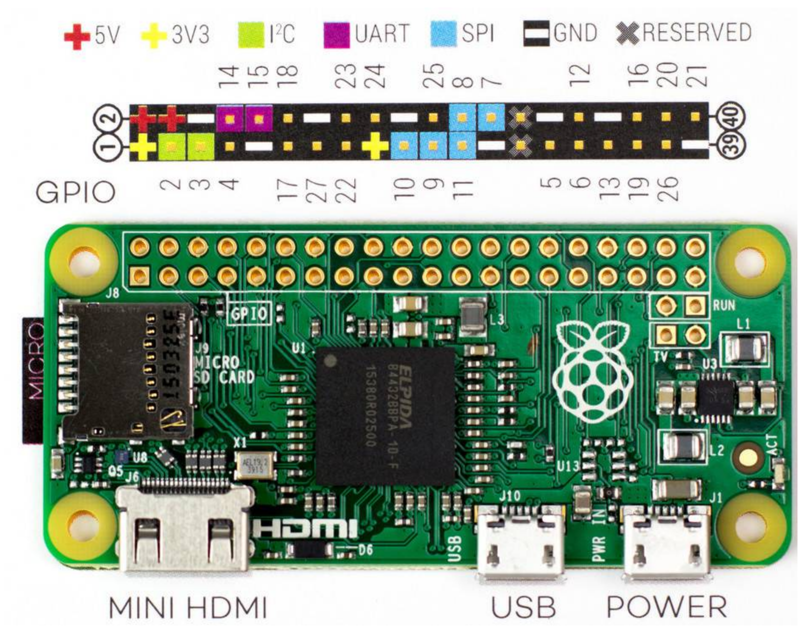


Illustration 4: GPIO models B +, 2, 3 and Zero.

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

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

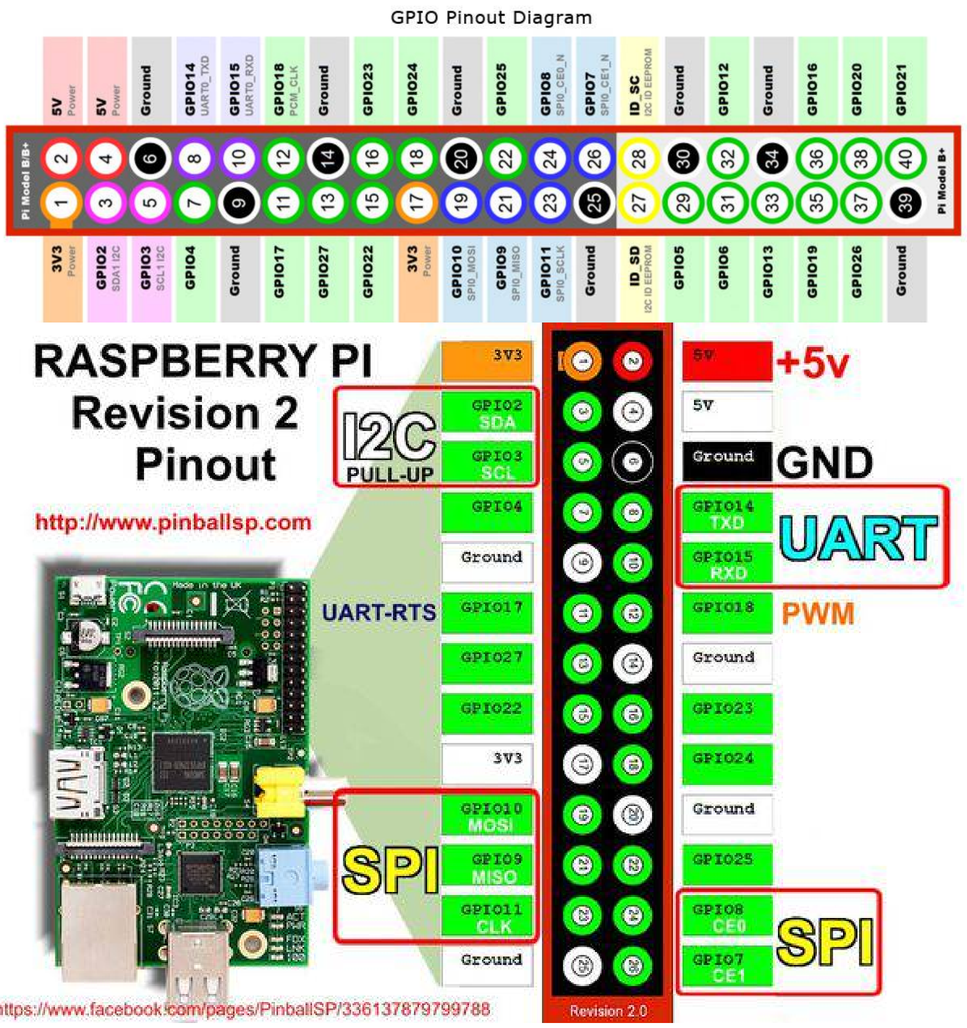


Ilustração 6: GPIO do modelo Raspberry Pi original.