MSXPi Interface Version 1.1
Ronivon Costa
ronivon.costa@gmail.com

Technical Reference Manual

Table of Contents

MSXPi Interface v1.0 is a design crearted and implemented by Ronivon Costa, 2016 – 2020.

1 MSXPi Overview

MSXPi is a hardware and software solution to allow MSX computers user Raspberry Pi devices as generic peripherals. I say “solution” because the hardware needs a paired Client/Server software in addition to the interface, to provide any useful functionality.

MSXPi hardware is composed of a MSX compatible interface to connect to the slot cartridge of any MSX 1,2,2+ and Turbo R. Due to some specific hardware implementation of some MSX models, the interface may not work on all MSX models. Also, the integrated MSX-DOS only works on 64K machines.

MSXPi software is composed of different components running on MSX and on Raspberry Pi. On the Raspberry Pi side, there is the msxpi-server component which listen for GPIO events, decode serial data using a few pins and transform that data into control commands and parameters. On the MSX side, there are different software components depending on how the user wants to use MSXPi.

- MSX-DOS is stored in the MSXPi EPROM. This is the MSX-DOS v1.0 customized with the MSXPi low level drivers to allow boot from MSX standard disk images stored on Raspberry Pi. The EPROM also contain a set of CALL commands to use from BASIC.
- MSX-DOS commands - these are “.com” commands that implement a series of functionalities on MSX-DOS allowing greater experience with MSXPi. This is the current approach to develop software for MSXPi, since it covers most of the users abse and allows for greater flexibility.
- MSXPIEXT.BIN with CALL commands to extend BASIC to access MSXPi without the need of the EPROM. This program adds a set of CALL commands in the RAM area in $4000, allowing new commands to be installed from DISK runnning “msxpiext.bin”. It actually installs the MSXPi BIOS into a simulated ROM cartridge installed in RAM.

In the next sections we will discuss the resources currently available for MSXPi, and how to use them.

1.1 What's new in version 1.1

This version is a major overhaul of the MSXPi. It was added support to the Z80 /wait signal, allowing a change to the low level drivers to be simpler, more straighforward and easy to support and add software. To support this major change, existing interfaces will need a small simple hardware mod, and to use the new software versions available.

Changes on Version 1.1

- PCB redesigned for software erasable EEPROM AT28C256. With this change the ROM can be re-written directly form the MSX-DOS using the included programmer. Jumpers modified to support the new features.
- Transfer routines changed again, for simplicity and stability. Uodated all clients and MSX-DOS driver.
- Pcopy command improved to detect when copies are done to the disk images, and copy the files directly to the image without the need to send to MSX, which in turn would send it back to the image.
- All “p” commands return a header: Return Code (1byte), Block size (2 bytes), Data (BLKSIZE). This is an optional parameter in the transfer function, which can be disabled for custom developed commands
- Added mapping of network & internet locations configurable by user using pset command: R1: (defaults to msx1 roms in msxarchive.nl), R2: (defaults to msx2 roms in msxarchive.nl), M: (defaults to local ftp server ftp://192.168.1.100). This work with commands pcd and pcopy, for example: pcopy /z R1:frogger.zip
- Added supported to automatically decompress archive files during the transfer with “pcopy /z”
- Added improved recover logic to stop MSXPi from staying in sync error loop.
- Lots of other bug fixes and improvements

Changes on Version 1.0

- PCB redesigned with support for /wait signal, use of by-pass capacitors for greater stability, and external pull-up resistors on all RPi GPIOs used by the interface.
- Requires a Mod to the existing interfaces v0.7: remove the BUSDIR jumper. Wire the CPLD pin 11 to the MSX /wait signal (requires a simple soldering skills).
- CPLD logic redesigned to implement the /wait signal, although at this version it will be set to tristate at all times.
- Server architecture completed re-written to be simple, more modular, easier to extend and maintain.

- All client applications re-engineered to support the new interface architecture.
- MSX-DOS ROM was recompiled to support the new interface architecture.
- Added CRC16 error correction on all download transfers (RPi to MSX)
- Server-side configurable number of transfers retry upon errors

Changes from Version 0.8.2

- A more complete set of CALL commands

  Starting at ROM build 20171230.00077, MSXPi contain a new set of CALL commands in ROM allowing it to be used in BASIC and from within BASIC programs.

  Not all MSXPi commands are be compatible, but some are (such as PRUN,PSET,PDIR) and allow exhange of data with Pi in BASIC.
- The new CALL commands also available as a BASIC extension (msxpiext.bin) for MSXPi that interfaces with older ROMs.
- New server in Python

  The msxpi-server has been ported to Python. This improves development time, at the same time allowing same level of transfer rate since the transfer of blocks of data are still using the C function.
- IRC client and WhatsUp client

  Two messaging clients are available, in BASIC, for these messaging platforms. They use the new CALL comands, and are compatible with either the new ROM or the msxpietx.bin extension.

2 Users Guide

This release comes with an EEPROM containing MSX-DOS 1.03. This is an unmodified MSX-DOS OS with drivers to support the MSXPi booting from a DSK disk image stored in the Raspberry Pi filesystem.
Note that there are structural changes to the server and client components. Because of that, the EPROM software (BIOS) and P commands must match the Raspberry Pi server component, or some commands will fail. Do not try to boot using old versions of the EEPROM or use old P commands with this release – they will not work.

MSXPi boots into MSX-DOS on MSXPi drive A: which is mapped to the disk image in /home/pi/msxpi/disks/msxpiboot.dsk. The MSX can also boot from MSX-DOS2 in another interface, and still have access to MSXPi files on the Raspberry Pi using P commands.

Drive B: is mapped to /home/pi/msxpi/disks/msxpitools.dsk which contains all currently available DOS commands to use with MSXPi and some other tools.

There are other three virtual devices mapped to be used by commands PCD, PDIR and PCOPY:

M: Mapped to ftp://192.168.100
R1: Mapped to http://www.msxarchive.nl/pub/msx/games/roms/msx1
R2: Mapped to http://www.msxarchive.nl/pub/msx/games/roms/msx2

All these mappings can be changed by the user to point to any other remote locaiton using PSET command.

When booting from MSXPi, you can also have a second disk drive interface attached (such as ATA-IDE or MegaFlashRom SD). These drives will be accessible from MSX-DOS1 and you can exchange files between MSXPi disk images and those drives (* Note: this is not currently working with Zemmix).

The MSXPi tools and how to use them is covered in a later section.

2.1 MSXPi ROM

The EEPROM contain the MSX-DOS 1.03, allowing MSX to boot into MSX-DOS directly from the interface. It contains also the BIOS (CALL commands) available to the user from BASIC:

- CALL MSXPIVER
  Display the ROM version and available commands

- CALL MSXPI
  Run MSXPi commands on RPi. Can optionally use a buffer to store data returned by the command.
- CALL MSXPISEND

Send any data to RPi, which can be a command, data, or both.

- CALL MSXPIRECV
  Receive data from Pi.

See the description and usage of these commands in 2.3 below.

2.2 MSXPIEXT.BIN & MSX-DOS ROM

MSXPIEXT.BIN is an extension for BASIC, that can be used by older MSXPi which ROM was not updated to latest releases. It can also be used without the need to have an EEPROM enabled in the MSXPi, which is the case if you only want to use MSXPi resources along with your MSX-DOS 2 or Nextor booting from other interface. This command should be loaded and executed to add new CALL commands to MSX-BASIC, and will only work on 64KB computers because it install itself as a ROM cartridge in RAM area starting at address #4000.

To install the extension, run the following command from BASIC:

bload"msxpiext.bin",r

A short help is diplayed with the available commands.

2.3 MSXPi CALL Commands

These commands use a buffer to exchange data with RPi. The buffer has a fixed length of 256 bytes (it's defined in the source code (include files) in the BLKSIZE constant. T

`CALL MSXPI(`*`"<stdout>,<buffer address>,<command>"`*`)`

Executes a command defined in the "*msxpi-server.py*", receives the reply data and process it according to the option specified in the "**stdout**" parameter.

Possible values for stdout parameters

stdout	Description
0	Ignore the date returned by RPi
1	Print tall received data o screen. If the data is 1 block only, full data is also available in the buffer. If it is larger than one block, only last block is available in the buffer
2	Store all received data in the buffer area. Note that if the data received is larger that the area reserved for buffer, there will memory corruption and possible crash

Buffer_Address: This is a string composed of four hexadecimal digits for a memory area to receive data from RPi – for example, "C000". It must have at least 256 bytes (BLKSIZE) available from this address.

Command: Any command defined in the “msxpi-server-py” program. Each command is created as a function in the server, as for example “pdir”, “pcd”, “prun” or any other command created by the developer.

_MSXPI uses a buffer of size BLKSIZE (256 bytes) to receive data from RPi. The first block received will contain a header, as shown in this table:

Address	**Buf+0**	**Buf+1**	**Buf+2**	**Buf+3 up to Buf+3+Size**
Content	Return code	# Data Size (lsb)	# Data Size (msb)	Data (this area size=lsb+256*msb)

All blocks will always have the same size – the first block will contain 3 bytes less available for data, because the first three bytes are used for header. All remaining blocks will have only data (no headers).

A transfer occurs BLKSIZE bytes at a time, and it may have one or more blocks. If the total data to transfer is larger than the BLKSIZE, then more than one block transfer will be required - the CALL MSXPI command takes care of these, making sure all data is available for the user. However, if you implement your own commands in the server to be called by CALL MSXPI, then you must assure that the server contains the correct logic to send the blocks sequentially as described. Refer to one of the MSXPi server commands to understand how it can be done.

lsb and **msb** bytes contain the data size for the data in this block. This is required because even tough the transfers occur in blocks of fixed size (BLKSIZE), the actual user data might be less that a block – all remaining data in the block is always padded with zeros. It’s also needed in cases where the data is larger than one block – the header contain the size for the whole data across all blocks. The client program must control how much data is read from RPi based on this header.

The size for the useful data in the block can be calculated in BASIC as follows (assuming it was passed “C000” in the CALL MSXPI command):

```
SIZE=PEEK(&HC001)+256*PEEK(&HC002)
```

If the returned data is larger than 509 bytes, then it must be transferred using two blocks or more (first block = 3 bytes header + 509 bytes of data, next blocks = 512 bytes of data).

Note that this buffer structure is hard-coded in the MSXPi Bios commands, and needs to be fulfilled by the respective MSXPi Server commands running in the Raspberry pi. This structure is only required to be set at the server side by specific commands developed to interface with this CALL command (typically, “output” parameter set to “2” – see description of CALL MSXPI below).

The return codes in the first block can be one of:

`RC_TXERROR`	Connection error or checksum error after all retries. This error is generated by the MSX client software because the transfer failed
`RC_SUCCESS`	Operation successful – This code is returned by the RPi. if any data is expected, it is available in the data area of the buffer.
`RC_FAILED`	Operation failed – this code is returned by RPi. The data area in the buffer should contain the error message from RPi operation.

Other return codes are available for use in development – consult the include.asm file reference.

When the command is called, the following scenarios may develop:

1. Connection error: RPi did not receive the command or it failed the checksums. The return code will be `RC_TXERROR` (set by the MSX due to lack of communication with RPi)

2. RPi received the command, processed it but it failed: The return code will be set by RPi to `RC_FAILED` and there will be an error message in the buffer - some server-side programs might set to a different error code, it's up to the developer to define which error code they want to use.

3. RPi received the command, processed and it succeed. The return code is RC_SUCCESS, and there will be data available in the buffer.

`CALL MSXPISEND(“`*`buffer_address`*`”)`

`“buffer_address” is a four digit hexadecimal number.`

`Send contents of buffer to RPi (it uses the SENDDATA function from msxpi_bios.asm). This command uses the buffer format as specified previously. The First byte should be left unused, and the next two bytes should contain the size of the data to transfer.`

`CALL MSXPIRECV(“`*`buffer_address`*`”)`

`“buffer_address” is a four digit hexadecimal number.`

`Read data from RPi and store in the buffer (it uses the RECVDATABLOCK function from msxpi_bios.asm). After completing the transfer, the first byte will contain the return code and the next two bytes will contain the number of bytes received.`

2.4 Examples using CALL Commands

2.3.1 CALL MSXPI

```
call msxpi(“pdir”)

call msxpi(“0,pdir”)
call msxpi(“1,pdir”)

call msxpi(“2,D000,pdir”)
```

Note: in the last example, since we passed flg “2”, the output of the command is saved in the buffer starting at address #D000.

2.3.2 CALL MSXPISEND

In this example, we will send data (a simple “TST” command) to RPi and verify the return code received.

```
10 gosub 100 ' Send a command to RPi
20 rc=peek(&hD000): ? hex$(rc) ' Get Return Code
50 end
99 ' Send data to RPi – a text command in this case
100 POKE &HD000,0 ' This is the area reserved for the return code
110 POKE &HD001,3:POKE &HD002,0 ' This is the buffer size (LSB,MSB)
120 POKE &HD003,ASC('T')
130 POKE &HD004,ASC('S')
140 POKE &HD005,ASC('T')
150 CALL MSXPISEND(“D000”) ' Send command TST to Rpi
160 RETURN
```

2.3.3 CALL MSXPIRECV

This example receive data from RPi and print on screen. For this reason, is is convenient that the data is ascii in a valid range so it will be correcly displayed on screen. This same routine also works with any binary data.

```
10 gosub 100 ' Receive data from RPi
20 rc=peek(&hD000): ? hex$(rc) ' Get Return Code
30 if rc=&hE0 then gosub 200   ' If RC=RC_SUCCESS then print data
50 end
99 ' Receive some data from RPi
100 POKE &HD000,0 ' This is the area reserved for the return code
110 POKE &HD001,0:POKE &HD002,0 ' This is the area reserved for buffer
size
120 CALL MSXPIRECV(“D000”) ' Received data from RPi
130 RETURN
200 S=PEEK(&HD001)+256*PEEK(&HD002) ' Get size of buffer
210 FOR M = 0 TO S-1 ' Will read all bytes in the buffer
220 PRINT CHR$(PEEK(&HD003)+M); ' and print on screen.
230 NEXT M:RETURN
```

3 MSXPi Architecture and Behavioural Description

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

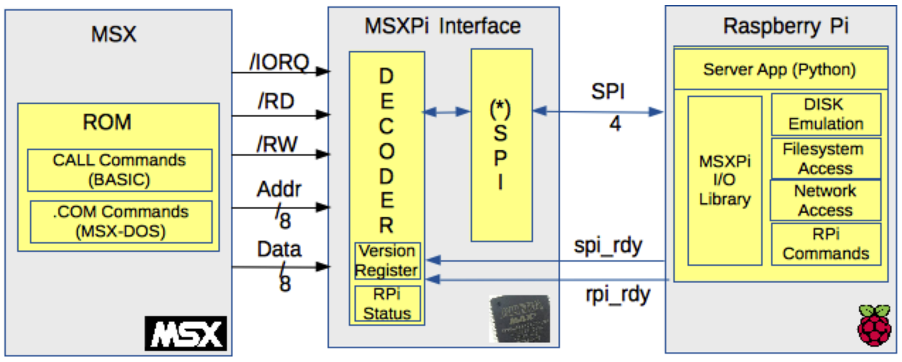


Figure 1: Architecture

1. MSX starts a transfer writing or reading I/O port 0x5a
2. MSXPi decode the address and signals, and decide if should enable MSXPi
3. MSXPi send /cs signal (low) to RPi, and /wait (low) to MSX
4. RPi Starts pulsing the SPI clock signal (this will trigger the MSXPi serial/paralell process)
6. MSXPi serial/paralell process will write or read the SPI signals, depending on the operation:
5.a For MSX OUT operation, MSXPi will serialize the MSX data and write to mosi pin
5.b For MSX IN operation, MSXPi will convert the bits from mosi pin to paralell and store in a latch
6. RPi will pusle signal spi_rdy to flag that the transfer is completed.
7. MSXPi will set the MSX data bus with the value received from Rpi (if a MSX IN operation)
8. MSXPi will release both RPi and MSX by:
8.a Setting /cs to HIGH
8.b Setting /wait to Tr-State ('Z')

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 specially because the server component is implemented in a slow, interpreted language (Python). Currently the benchmark transfer rate is approximately 5KB per second (Kilo Bytes / s).

The interface between MSX and RPi is made using a CPLD tolerant to 5 volts, the EPM3064 from Altera. In this CPLD it is implemented the logic to decode the I/O port and other MSX bus signals (/wr, /rd, /iorq) and transfer a byte between MSX and Raspberry Pi using a simple shift register clocked by the Raspberry Pi once the MSX writes or read to the MSXPi port. Due to the fact that the clock for the transfer is generated by the RPi, the speed is mandated by that device, which will be impacted by background applications running along with the msxpi-server.py component.

The client application s (.com commands) was developed in the Assembly Z80 language, and the server application Python. Performance can be improved by developing components in C language at the server-side, if needed.

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.

4 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	

When using RPi Zero, power is supplied by MSX.
When using RPi 1,2 or 3, RPi must be connected to an external power supply.

To attach Raspberry Pi Zero to the interface, use the photo in illustration 2 as a reference. As for software version 1.0, only 7 PINs are used on Raspberry Pi Zero as indicated with the red arrows. Soldering is only required in the signaled pins.

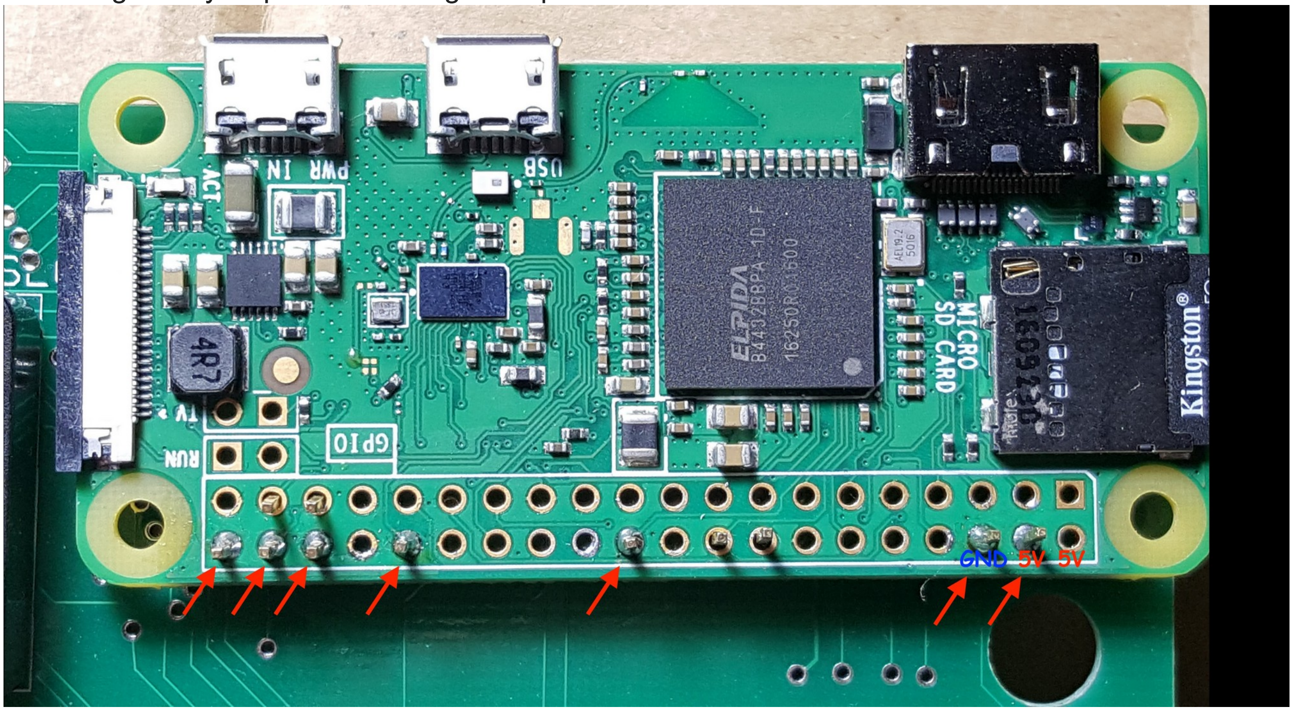
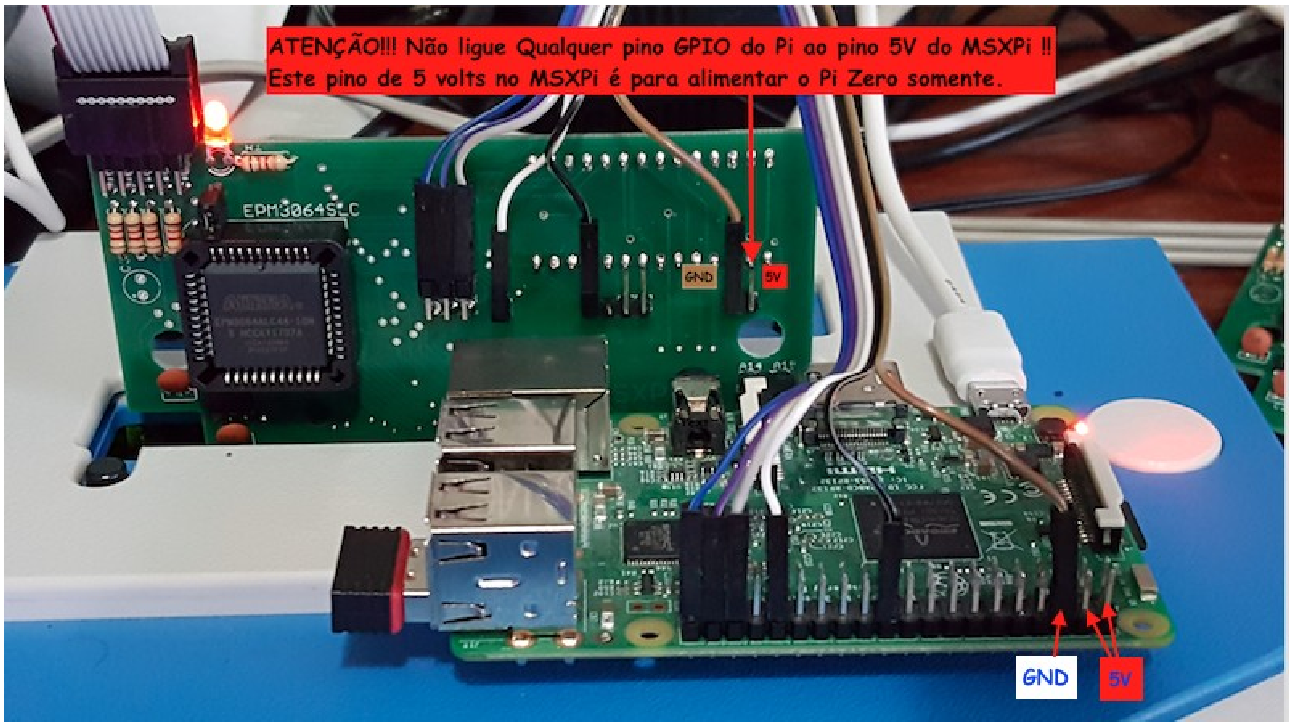


Figure 2: Attaching Raspberry Pi Zero to the interface

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


ATENÇÃO!!! Não ligue Qualquer pino GPIO do Pi ao pino 5V do MSXPi !!
Este pino de 5 volts no MSXPi é para alimentar o Pi Zero somente.
EPM3064SLC
GND
5V
GND
5V
Figure 3: Connection of the B +, 2 and 3 models

<u>**Note**</u>**: It is essential that the MSXPi GND (brown wire) is connected to the Raspberry Pi. Without this connection, the signals on the interface pins will not be transmitted correctly.**

5 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 and Python framework on the Raspberry Pi side. In this section we focus on the MSX-side library.

The MSXPi project has the following structure:

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

/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 functions in the libraries mentioned previously.

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

Next section will describe the bios functions only.

5.1 MSXPi BIOS (MSXPI_BIOS.ASM)

CHKSPIRDY

Label : CHKPIRDY

Purpose	: Read the MSXPi Interface status register (port 0x56)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 56h 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 5AH) 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 56h, then call CHKPIRDY to know when a byte is available to be read. In the sequence, the routine read data port 5Ah 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 CHKPIRDY to know when Pi is available to receive data, and in the sequence write the data to the data port 5Ah.
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 56h).
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.

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

RECVDATA
Label: RECVDATA
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

SENDDATA
Label: SENDDATA
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

Note: There are more useful routines available as part of the MSXPI BIOS. Read the msxpi_bios.asm source code for details. Useful routines are:

- LOADBINPROG
- PRINT
- PRINTDIGIT
- PRINTNUMBER
- PRINTPISTDOUT

- NOSTDOUT
- STRTOHEX
- CHECK_ESC

5.2 BIOS Functions Address Table in ROM

This table show the actual addresses for the MSXPi bios functions in the ROM. To use these functions, the slot where MSXPi is connected must be first identified, and then an interslot call can be made to the required address.

Note that the functions are not aware of slot switching and ram banks, which means that any buffer provided to the function must be directly accessible by the function.

Note: Address valid for ROM build **`20171230.00077.`**

Address	Function
75CC	MSXPIVER
75D2	MSXPISTATUS
75FA	MSXPILOAD
79D8	RECVDATABLOCK
7A0B	SENDDATABLOCK
7A44	SECRECVDATA
7A74	SECSENDDATA
7AA0	READDATASIZE
7AAD	SENDDATASIZE
7D50	CHKPIRDY
7D61	PIREADBYTE
7D6F	PIWRITEBYTE
7D77	PIEXCHANGEBYTE
7D4D	SENDIFCMD
79D4	SENDPICMD
7B6C	CHECKBUSY
7B86	PRINT
7BA4	PRINTNLINE
7BAF	PRINTNUMBER
7BC4	PRINTDIGIT

7BD3	PRINTPISTDOUT
7858	79AA
7AB6	DOWNLOADDATA
7BB6	UPLOADDATA
7703	MSXPISEND
772B	MSXPIRECV
7642	MSXPI

To identify the MSXPi slot, you need a search routine (not currently provided). Different methods can be used to find where MSXPi is connected, but you can use the following addresses and string patterns:

Address	String
780A	`MSXPi Hardware Interface v0.7`
782B	`MSXPi ROM v0.8.2`
7849	`<14 characters in this position contain the build id>`

5.3 MSXPi Server

Soon.

6 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 0x5A to get the byte.

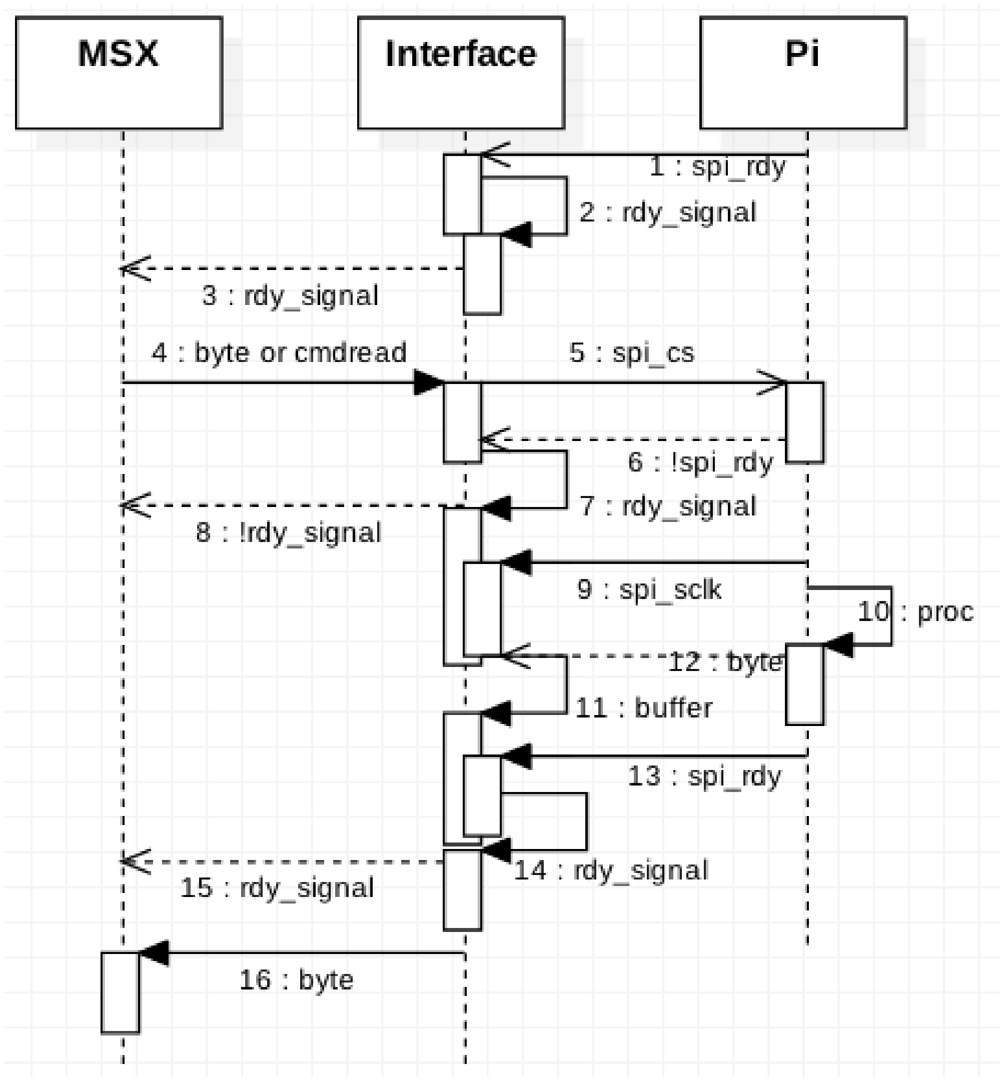


Illustration 2: Diagram Sequence for the Interface

7 Appendix

7.1 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.

7.2

For ROM-less MSXPi

MSXPi BIOS can be used by loading the msxpiext.bin from BASIC. This extension will load itself as a cartridge and implement all the BIOS routines and CALL commands found in the EEPROM. This extension needs a MSX with 64K of RAM.

7.3 Appendix 2: Pi - GPIO Pin Numbering

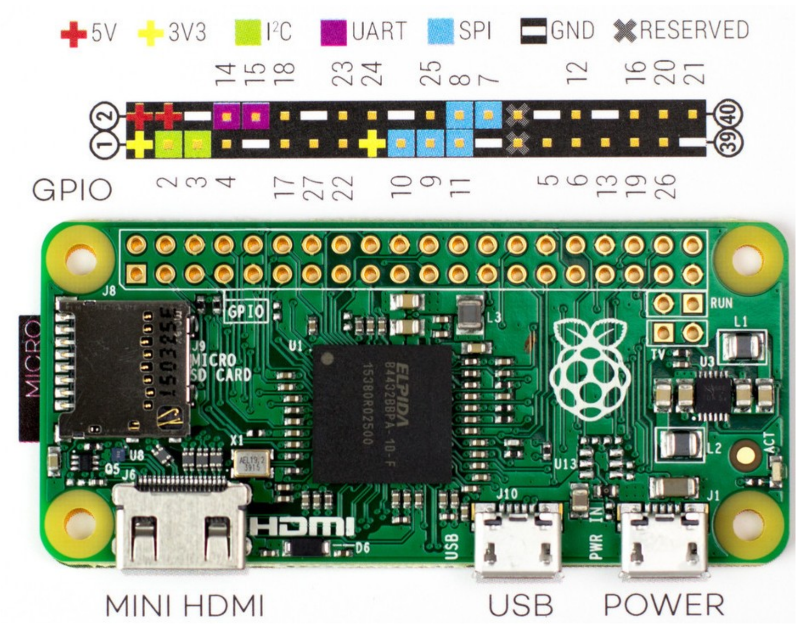


Illustration 4: GPIO models B +, 2, 3 and Zero.

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

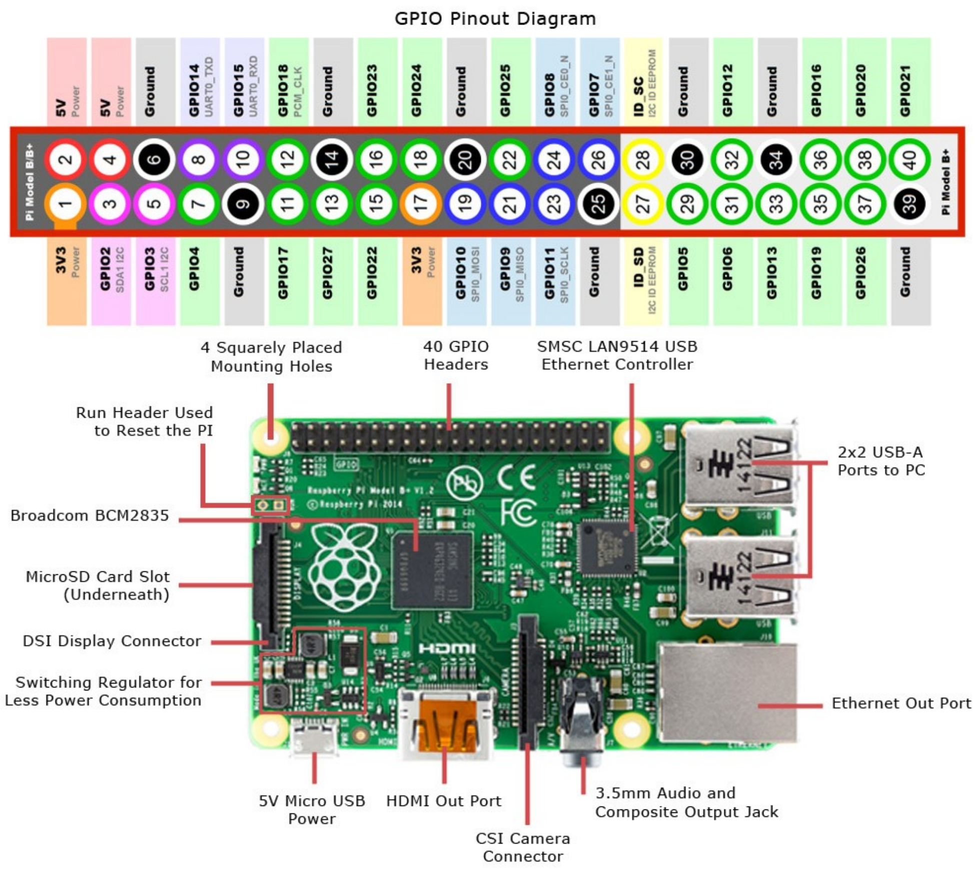


Illustration 5: GPIO models B +, 2, 3 and Zero.