MSXPi Interface Version 1.2

User's and Developers Guide

DRAFT – Not Revised

Table of Contents

1 Introduction

MSXPi is a hardware and software solution to allow MSX computers to use Raspberry Pi devices as generic peripherals. The project implements a MSX compatible hardware interface that can be used with most MSX 1,2,2+ and Turbo R. Due to some specific hardware implementation of some MSX, the interface may not work on all MSX models. Also the integrated MSX-DOS only works on 64K machines (this is a MSX-DOS requirement).

MSXPi requires a Raspberry Pi connected to the interface:

- Raspberry Pi Zero W

  This is the recommended Raspberry Pi model, because the form factor is ideal to plug into the interface, and the low power consumption allows it to be powered by the MSX without the need for another external power supply.

  The fact that it also has wireless networking makes it the ideal Raspberry Pi for MSXPi.

- Raspberry Pi Zero

  The second best Raspberry Pi model to plug into MSXPi. This model comes without the WiFi module, but has all other capabilities of the model above. To use networking features of MSXPi, a WiFi dongle need to be connected in the USB port.

- Raspberry Pi Mode 2/3/4

  These models can also be used with MSXPi, although the connection will be awkward with the use of wires and external power supply. Not recommended for daily use, although can be use for development, experimentation and tests. One advantage of this model is that it is faster, and has audio output that can be used along with pplay command.

MSXPi cannot be used with any Raspberry Pico models, as they are in fact micro controllers and do not run Linux as their base operating system, which is required for the MSXPi server program.

MSXPi software is composed of different components running on MSX and on the Raspberry Pi. The Raspberry Pi run the server component which listen for GPIO events, decode serial bits using a four pins and transform them into commands and data. In the MSX side, there are different software components depending on how the user wants to use MSXPi:

- MSX-DOS ROM is stored in the MSXPi EEPROM. This is the MSX-DOS 1 customized with the MSXPi low level drivers to allow the MSX to boot from disk images stored in Raspberry Pi SD card. The ROM also contain the MSXPi BIOS with a set of CALL commands that can be used from BASIC.

- MSX-DOS commands - these are the “P” commands that implement a series of functionalities in MSX-DOS allowing greater experience with MSXPi, such as setting system clock and access to internet resources form MSX-DOS 2 or Nextor.
- MSXPIEXT.BIN is the MSXPi BIOS implementation that can be loaded from BASIC and behave similar to a MSXPi with an installed ROM. The BIOS is installed in the RAM in the 4000h cartridge area, becoming available for BASIC programs.

1.1. What's New

This version is a major overhaul of the MSXPi The most significant changes are listed next.

Changes in Version 1.2

- Dropped the reserved 8 I/O ports in range 0x56 – 0x5D, changing the BIOS to use only ports 0x56 (control) and port 0x57 (data). Existing users that want to upgrade to this version will have to program the CPLD with the firmware version 1.2, or recompile the MSXPi project locally after reverting the DATA_PORT1 to $5AH in MSXPi\software\asm-common\include\include.asm
- Added a new ROM (msxpibios.rom) with BIOS for BASIC (CALL commands from BASIC)
- MSX-DOS 1.3 booting from a disk image in Raspberry Pi is now discontinued., It stays available as source code and ready to use ROM, but for research purposes only. No support or further development will be provided for this component.
- A MSXPi Extension was added for openMSX, with support for all MSXPi commands. The MSXPi commands should work on both the physical MSX + MSXPi and emulated MSX + MSXPi inside openMSX. The Raspberry Pi is not required, as the msxpi-server.py will be run in the local host where openMSX is running. Consult the Appendices for details of the MSXPi extension for openMSX.

Changes in Version 1.1

- PCB redesigned to use EEPROM AT28C256. With this change the ROM can be re-written directly form the MSX-DOS using the included programmer. Jumpers were expanded to support the new feature.
- BIOS transfer routines changed, for simplicity and stability. Updated all client software and MSX-DOS drivers.
- Pcopy command improved to detect when destination drive is a disk image, and copy the files directly to the image on the Raspberry Pi side without the need to use MSX-DOS read/write, greatly improving performance when copying files and saving internet files directly to the disk image.
- All “P” commands returns a header in the data: Return Code (1 byte), Block size (2 bytes), Data (BLKSIZE). Headers are an optional parameter in the python 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 R1:frogger.zip
- Added support decompressed archive files during the transfer with "pcopy /z". When using "/z" parameter, the file is decompressed and only the decompressed file is saved in the disk.
- Improved the server component recovery logic to stop MSXPi from going out of sync sync indefinitely. Recovery times takes no more than five seconds and it is automatic.
- Lots of other bug fixes and improvements

Changes in 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 Raspberry Pi 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 (Raspberry Pi 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 exchange 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 commands, and are compatible with either the new ROM or the msxpiext.bin extension.

2 Getting Started

MSXPi can be used with or without the EEPROM in the interface. In fact, the best way to use MSXPi is to use it along with another IDE interface running MSX-DOS 2 or Nextor, with MSXPi being used as a Raspberry Pi controller and network interface. The EEPROM can be removed (if socketed) or disabled by removing the SLSLT jumper.

Note: *MSXPi networking protocol is proprietary and does not support TCPIP. Therefore programs that require TCPIP interface will not detect or work with MSXPi.*

The remaining of this guide may mention “P” commands, BIOS, ROM, EEPROM programmer and other software – all of these should be downloaded from the MSXPi repository in github and saved to the MSX disk or SD Card when using MSX-DOS2 or Nextor.

Dot it now: As generic recommendation, have the most recent MSXPi commands and files in your MSX disk or SD card, by downloading them from github repository – these files are in the /software/target folder.

The MSXPi interface has the following main components:

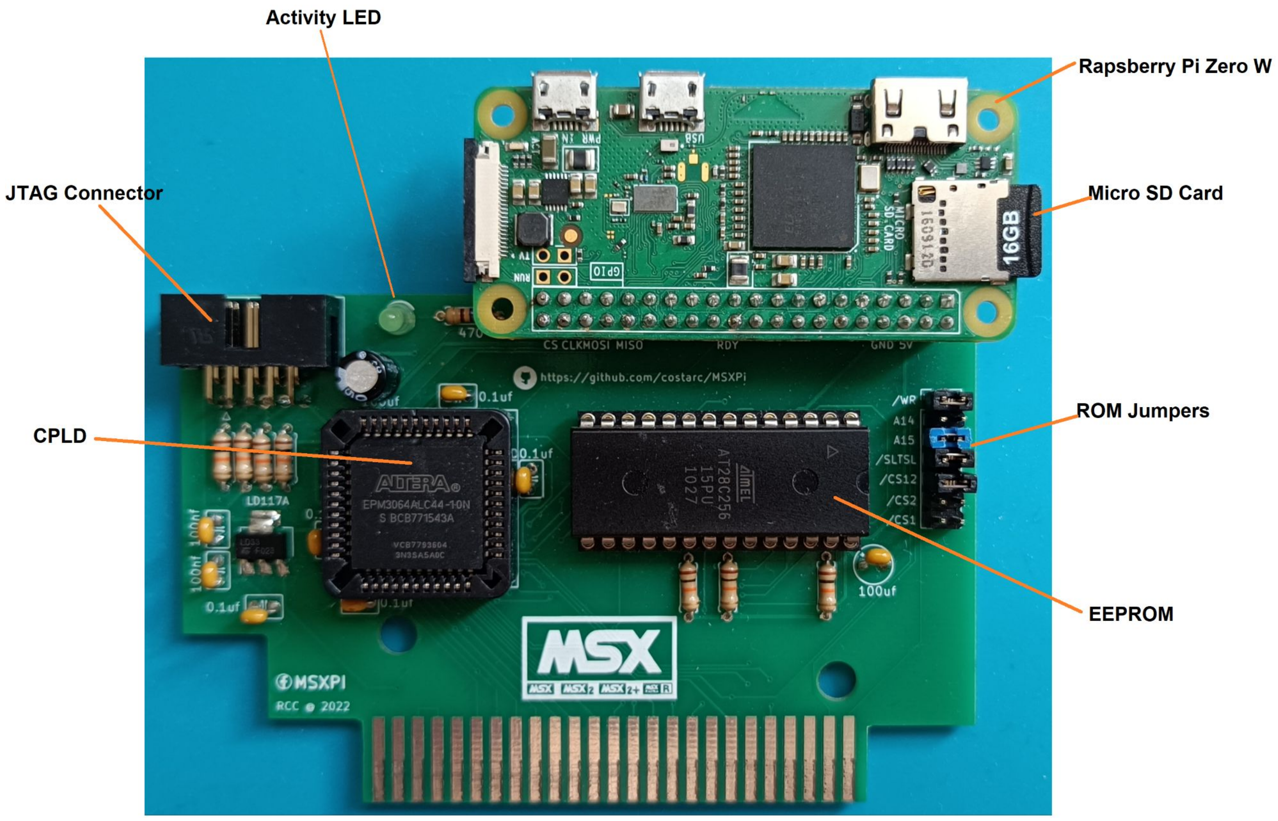


Illustration 1: MSXPi Interface Components

- Raspberry Pi Zero W

  This is the essential component in the interface. It boots into Linux, and starts up the msxpi-server.py application that implements all MSXPi commands on the server side.

- Micro SD Card

This SD Card is used by the Raspberry Pi to boot into Linux. There is a pre-configured image in the MSXPi repository containing a recent Raspbian OS and all the required MSXPi programs and libraries.

- EEPROM AT28C256

  This EEPROM contains the MSXPi BIOS and MSX-DOS drivers required to boot into MSX-DOS 1.03 from a disk image stored in the Pi SD Card. This EEPROM can also be used for other purposes, such as storing ROM games. This EEPROM should be written from MSX-DOS using the supplied AT28C256.COM programmer.

- ROM Jumpers

  Used to disable the ROM and also select one of the two 16KB ROM banks available.

- CPLD

  Contains the firmware to decode I/O ports used by MSXPi, and do all data transfers between MSX and Raspberry Pi.

- JTAG Connector

  USB-Blaster compatible connector used to program the CPLD with new firmware versions or other custom firmware. CPLD programming cannot be done via MSX, and requires the Altera software installed in a Windows computer. To program the CPLD, the interface must be inserted into the MSX and the MSX turned on.

- Activity LED

  Flashes whenever the interface is communicating with the Raspberry Pi

2.1. Assembling the Interface

When buying the Raspberry Pi for MSXPi, verify how the Raspberry Pi pin socket is soldered in the MSXPi PCB. If the socket is soldered in the back of the PCB, the Raspberry Pi Zero WH (with pre-soldered pin headers) will fit without changes or soldering. If the socket is in the front of the MSXPi PCB, the Raspberry Pi Zero WH cannot be used, and the user will be required to solder pin headers in the Raspberry Pi as in the illustration 2 below.

A 2x20 pin header must be soldered to the Raspberry Pi facing down, in order to safely connect it to the MSXPi interface. Refer to the following images to solder the pin header in the Raspberry Pi Zero and plug it to MSXPi.

Note: *The Raspberry Pi must always be plugged into the MSXPi facing the user. This rule is true for any Raspberry Pi version: Zero, Zero W and Zero WH - Refer to Illustrations 5 and 6 below.*

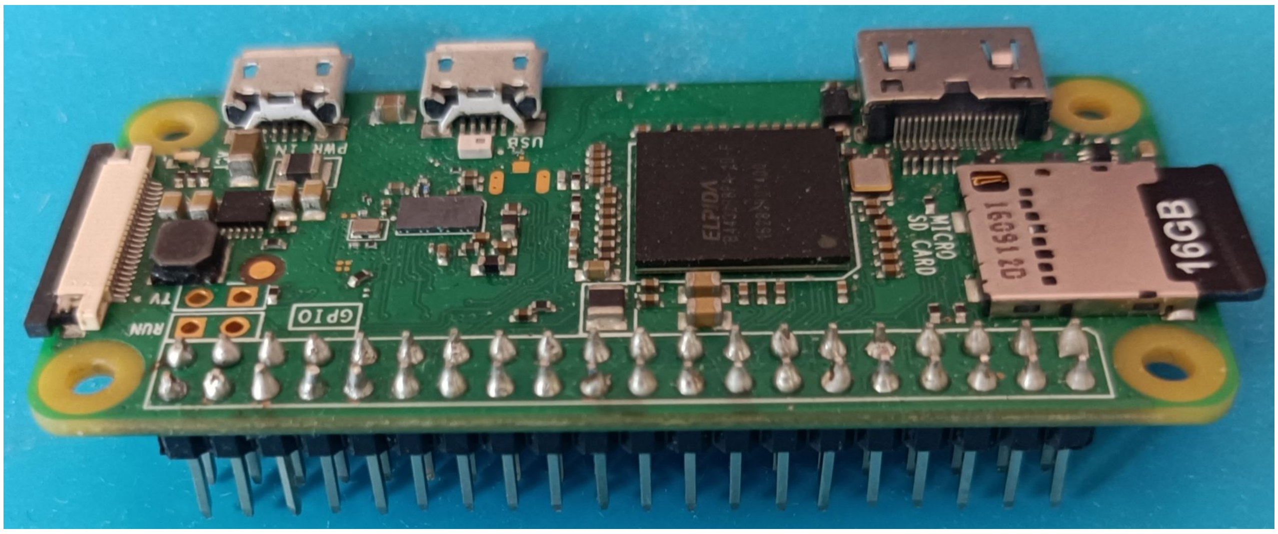


Illustration 2: Header Install – Top View

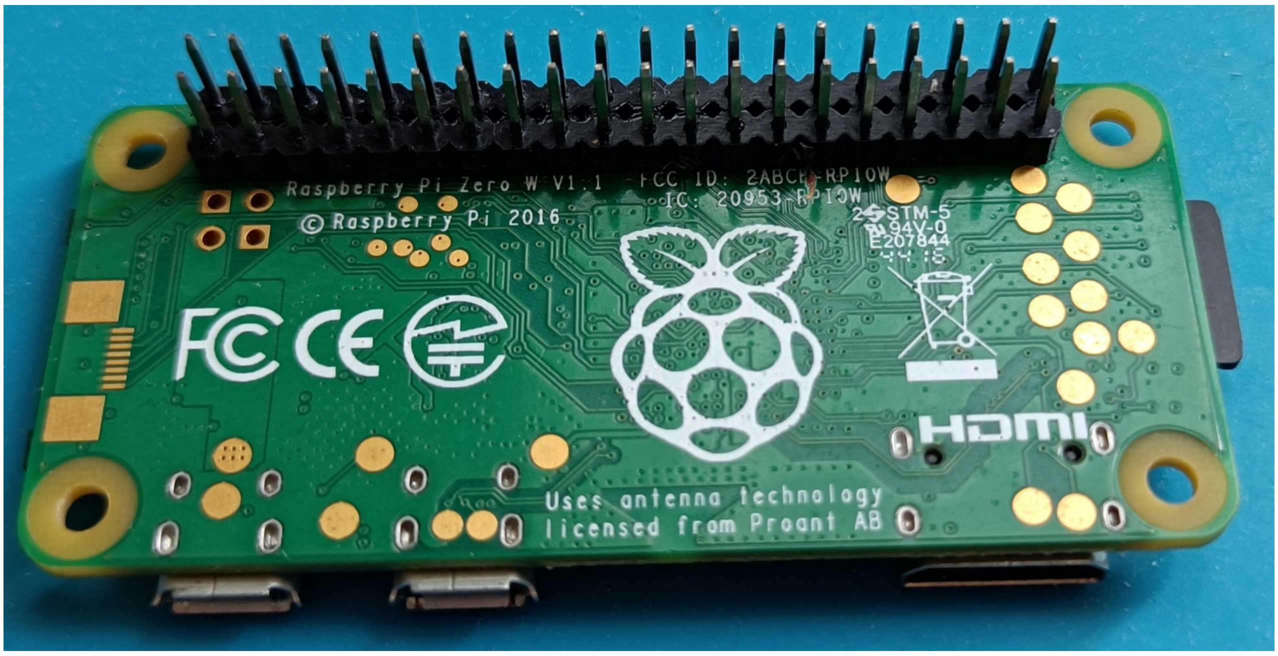
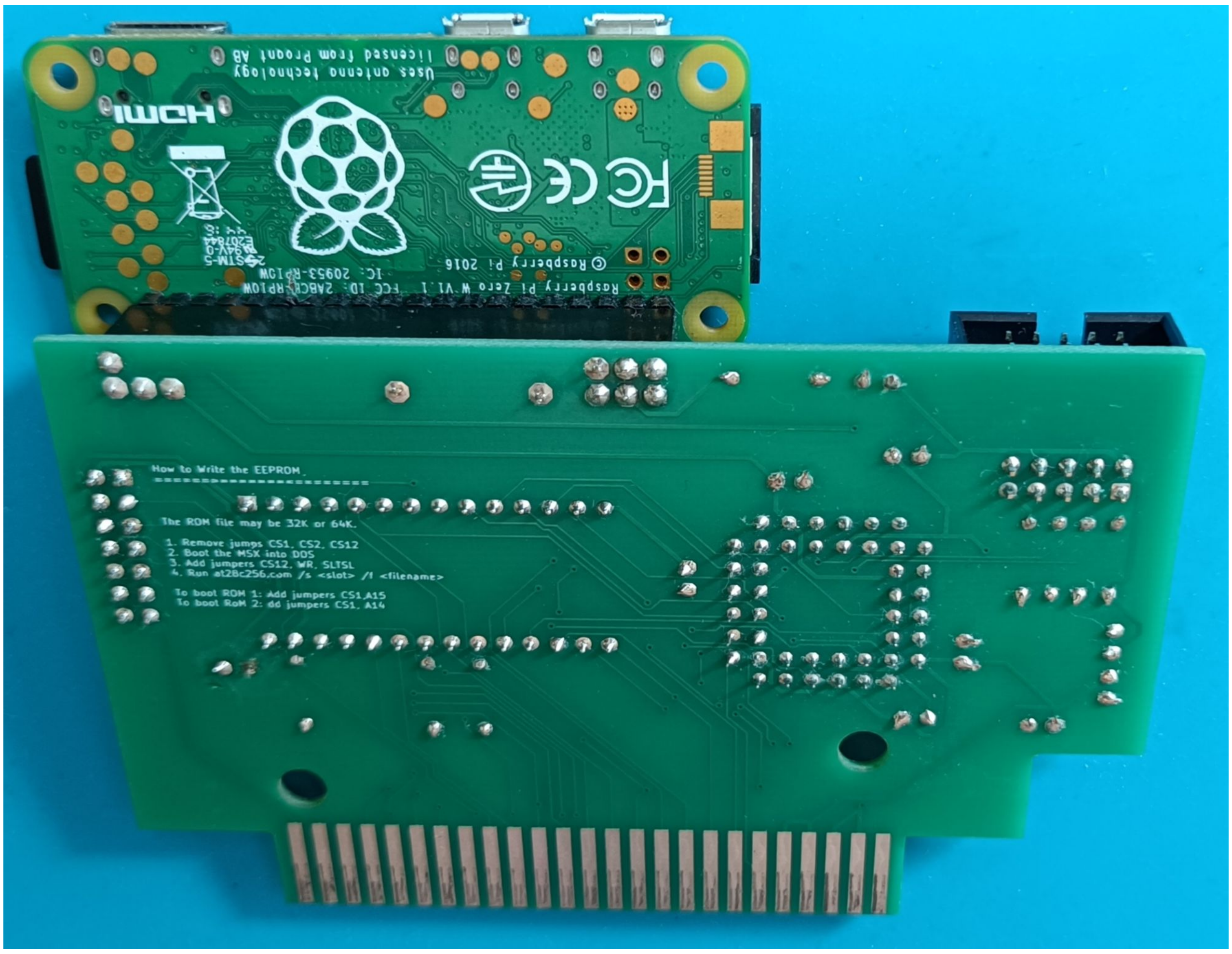


Illustration 3: Header Install – Bottom View


How to Write the EEPROM
=========================

The ROM file may be 32K or 64K.

1. Remove jumgs CS1, CS2, CS12
2. Boot the MSX into DOS
3. Add jumpers CS12, WR, SLTSL
4. Run at28c256.com /s <slot> /f <filename>

To boot ROM 1: Add jumpers CS1,A15
To boot RoM 2: dd jumpers CS1, A14

Illustration 4: MSXPi Assembled - Bottom View

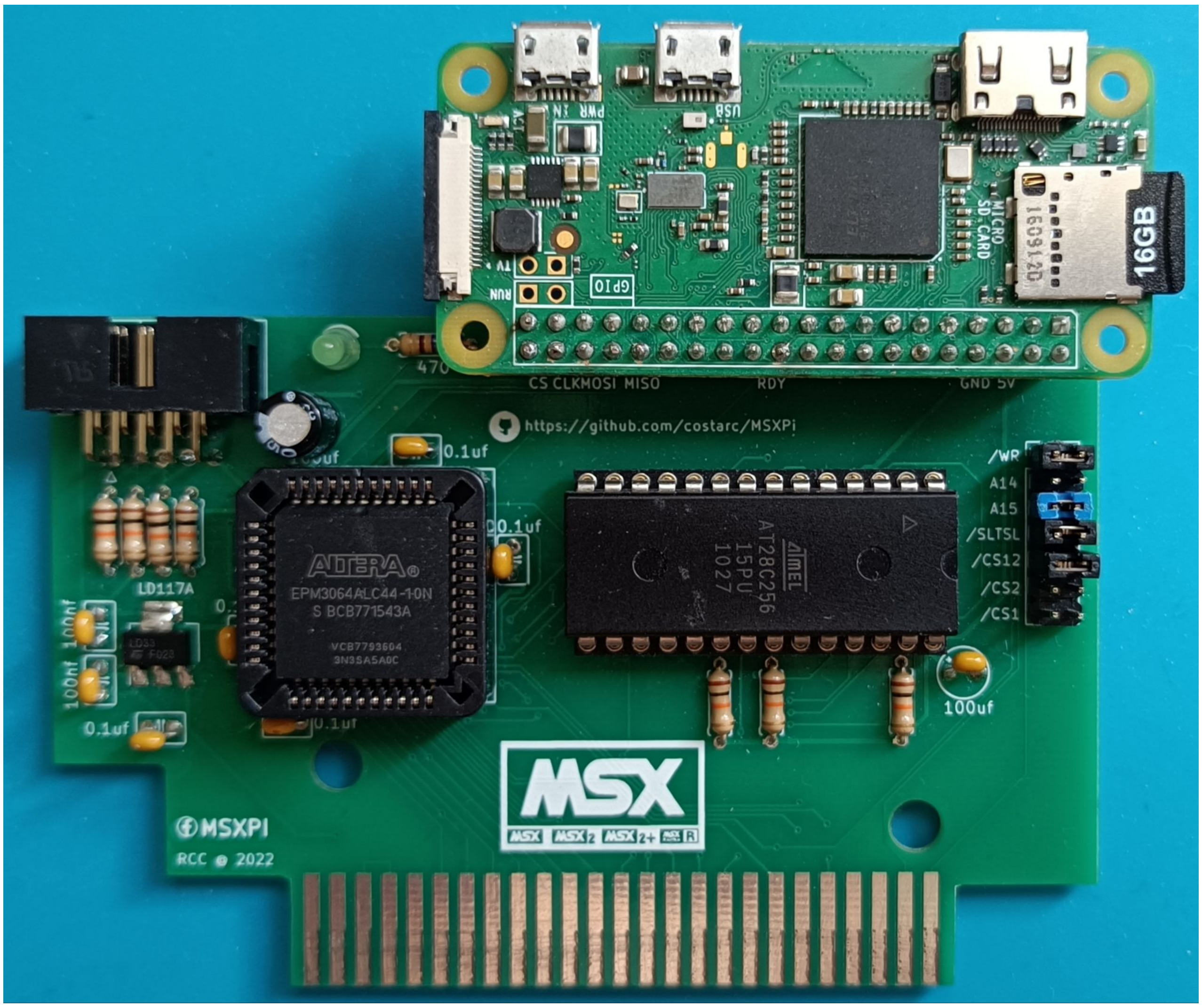


Illustration 5: MSXPi Assembled - Top View

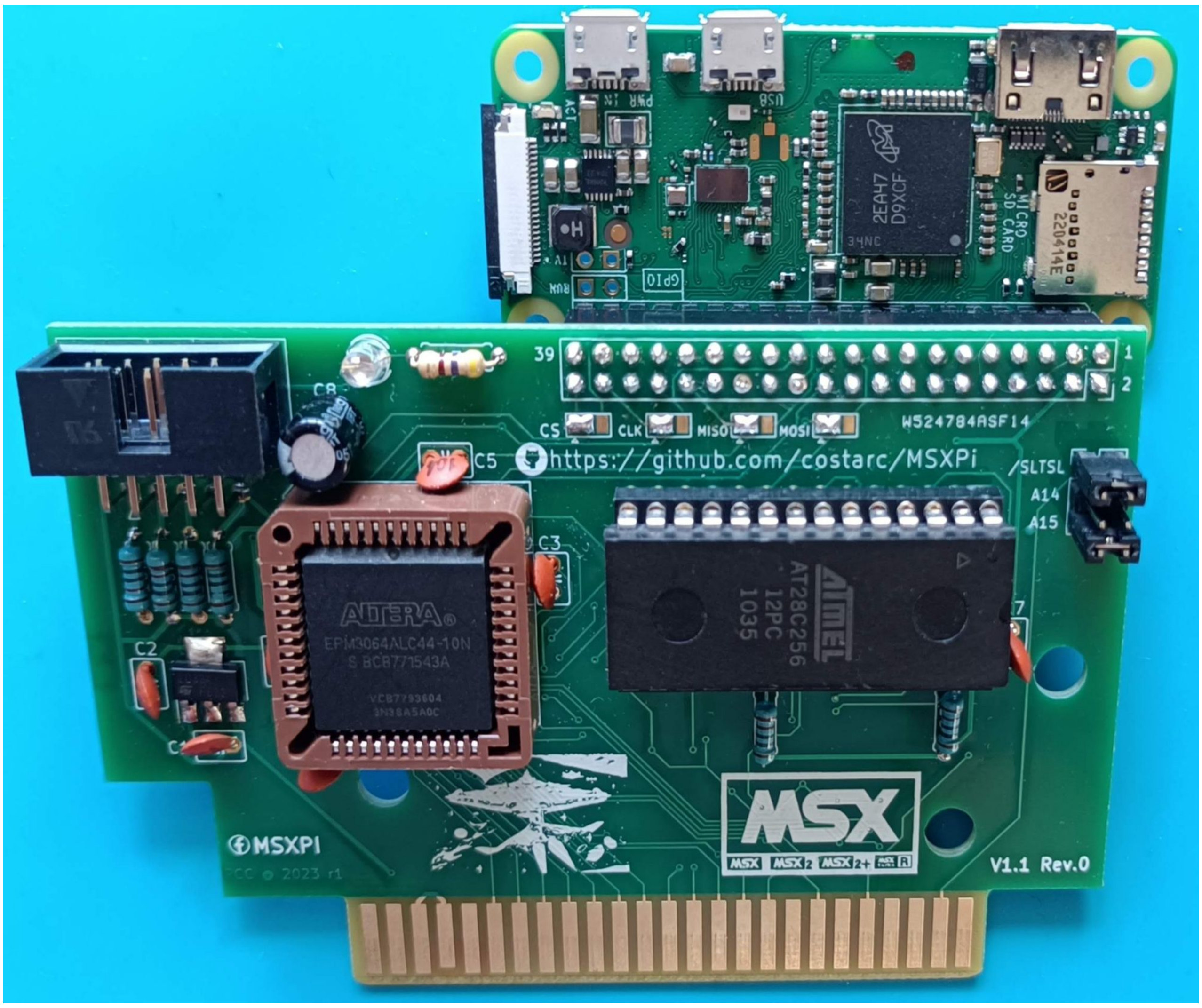


Illustration 6: MSXPi Assembled - Top View

2.2. Preparing the SD Card for Raspberry Pi

This Quick Start Guide is updated to V1.1 of the Software and Interface.

Please refer to the full documentation under "documents" folder in github for detailed setup procedure and other information.

There are a few steps to setup MSXPi, and you can choose between using a MSXPi pre-installed SD Card image (ready to boot MSXPi), or build your own image using a fresh Raspbian Image downloaded from Raspberry Pi web site - both methods are described below.

Overall, the steps to get up and running are:

- Setup Raspberry Pi with the server-side software (Raspberry Pi SD Card)
- Setup MSX with the client side software (MSX SD Card / disk drive)

2.2.1 Using the MSXPi Pre-Installed SD Card

Step 1: Download MSXPi SD Card image.

MSXPi SD Card image: https://tinyurl.com/MSXPi-SDCard

Step 2: Write the image to a SD Card suing Pi Imager

Raspberry Pi Imager can be download from https://www.raspberrypi.com/news/raspberry-pi-imager-imaging-utility

Use a SD Card with a minimum of 4GB.

Use 7Zip to unzip the file, and use Raspberry PI Imager to write the image to the SD Card - in the Choose OS drop box, select "Use Custom" and pick the SD Card image downloaded in step 1.

Step 3: Install the MSXPi commands for MSX-DOS

In your favourite PC computer, copy all MSXPi commands from https://github.com/costarc/MSXPi/tree/master/software/target to your MSX SD card or Disk.

After this basic setup, you should be able to use the MSXPi "P" commands from your MSX. To unleash full MSXPi power, configure the Raspberry Pi Zero W Wi-Fi:

```
pset WIFISSID Your Wifi Name

pset WIFIPWD Your Wifi Password

pwifi set

preboot
```

Note: The first reboot may take longer than 3 minutes, because Raspbian will expand the filesystem in the SD and initialize the Linux system - following reboots will be faster.

In case you need very detailed instructions, please read "Tutorial - Setup Raspberry Pi for MSXPi the Easy Way - Using the MSXPi Pre-Installed Image.pdf", in https://github.com/costarc/MSXPi/tree/master/documents (Portuguese version also available).

2.2.2 Installing and configuring Raspbian from Scratch

In this mode, you will have to install all requirements for MSXPi - there is a script to help you with that, though.

Step 1: Download the Raspberry Pi Imager

Raspberry Pi Imager can be download from https://www.raspberrypi.com/news/raspberry-pi-imager-imaging-utility

This is the official Raspberry Pi SD Card image writer - download and install in your desktop PC.

Step 2: Write the Raspbian image to the SD Card

Run the Pi Imager software, and select the best OS for your raspberry pi. If you are using the recommended Raspberry Pi Zero W, choose the lite version (without graphical desktop):

```
CHOOSE OS -> Raspberry Pi OS Lite (other) -> Raspberry Pi OS LITE (32-bit)
```

Write the image to your SD Card and when finished, boot the Raspberry with the SD Card inserted.

Step 3: Setup MSXPi using MSXPI-Setup tool

You will need to connect the Raspberry Pi to a HDMI TV and a keyboard to complete these steps.

Login to Raspbian using default user and password: pi / raspberry

Configure the WiFi using raspi-config command

Download the MSXPi setup script - it will download and install everything needed to have MSXPi up and running:

```
mkdir /home/pi/msxpi

cd /home/pi/msxpi

wget https://tinyurl.com/MSXPi-Setup

chmod 755 MSXPi-Setup

sudo ./MSXPi-Setup
```

If you need very detailed instructions, please read "Tutorial - Setup Raspberry Pi for MSXPi the Hard Way - Installing Raspbian from Scratch.pdf", in https://github.com/costarc/MSXPi/tree/master/documents (Portuguese version also available).

2.3. Booting from Another Interface with MSX-DOS 2/Nextor

This is the recommended method to use the MSXPi, because you use a primary IDE interface with MSX-DOS 2 or Nextor at same time using the MSXPi as a secondary interface to control the Raspberry Pi and access network resources.

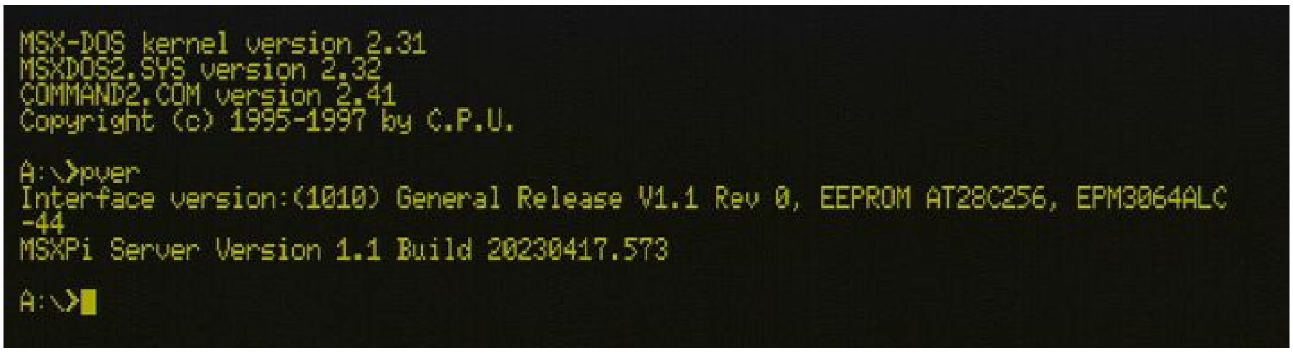
Remove Jumper SLTSL – this will prevent the MSX from booting from the MSXPi ROM.

Plug the MSXPi with the prepared SD Card inserted in the Raspberry Pi Zero W.

Switch on the MSX, it should boot straight into the MSX-DOS 2 or Nextor from your IDE interface - this boot speed won't be impacted by the Raspberry Pi boot, however, the MSXPi will not be immediately available until the Raspberry Pi boot is completed and the MSXPi server initiated.

At any time, check if the MSXPi is available using the "pver" command. Use ESC to interrupt and try again after five seconds if the MSX seems to become unresponsive.

Once the MSXPi boot is complete, you have a MSX-DOS 2 or Nextor environment to play with along all the features of the MSXPi by using the "P" commands.


```
MSX-DOS kernel version 2.31
MSXDOS2.SYS version 2.32
COMMAND2.COM version 2.41
Copyright (c) 1995-1997 by C.P.U.

A:\>pver
Interface version:(1010) General Release V1.1 Rev 0, EEPROM AT28C256, EPM3064ALC
-44
MSXPi Server Version 1.1 Build 20230417.573

A:\>
```

2.4. Booting from MSXPi ROM

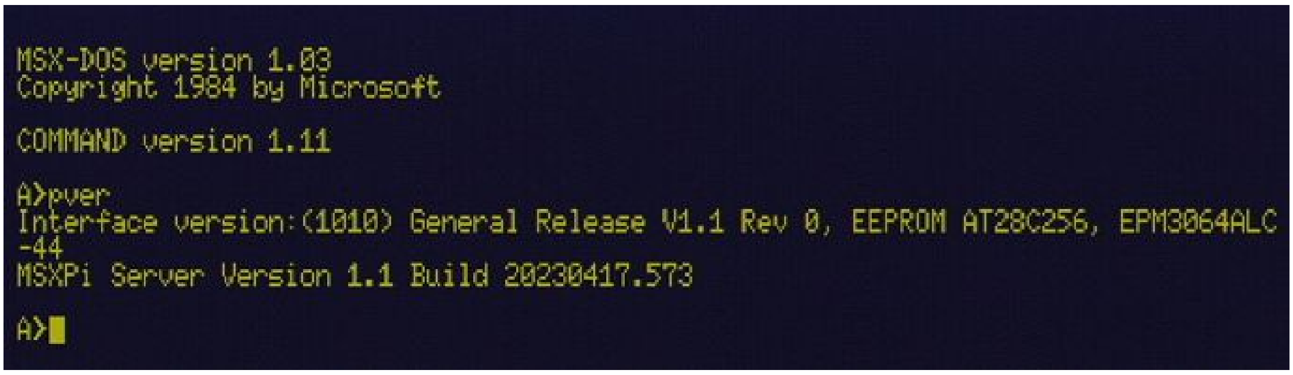
Make sure the ROM contain the most recent MSXPIBIOS.ROM. If necessary, download it from github and update with AT28C256.COM programming software – this should be done from a working MSX with MSX-DOS or Nextor.

Close jumpers A15 and SLTSL – this will make MSX boot from the MSXPi ROM.

Plug the MSXPi with the prepared SD Card inserted in the Raspberry Pi Zero W.

Switch on the MSX, and wait until the MSX-DOS 1.03 messages appear – because the MSX is booting from a disk image in the Raspberry Pi, the first boot takes at least couple of minutes.

Once the boot is complete, you have a MSX-DOS 1 environment available. This Drive A: and B: are mapped to disk images in the Raspberry Pi.


```
MSX-DOS version 1.03
Copyright 1984 by Microsoft

COMMAND version 1.11

A>pver
Interface version:(1010) General Release V1.1 Rev 0, EEPROM AT28C256, EPM3064ALC
-44
MSXPi Server Version 1.1 Build 20230417.573

A>
```

Do not delete files in drive A:, because they are needed for the MSXPi to boot and operate correctly in this mode.

3 Users Guide

MSXPi can be used as a stand-alone floppy disk drive, booting from a MSX-DOS 1.03 in a disk image (.dsk) stored in the Raspberry Pi. This works to an extent, although there are several limitations and bugs. It’s not the recommended way to use the MSXPi, although it is a reasonable option if there is not another storage device available to boot the MSX with Nextor or MSX-DOS 2.

The best method to use MSXPi is to use it along with the SD Card interface running Nextor or MSX-DOS2 (such as a Megaflashrom SCC+ SD). In this configuration, the user has the best option for boot and storage in parallel with all networking features provided by MSXPi.

Either way, the "P" commands are available and can be used in the MSX-DOS 1, 2 and Nextor. The MSXPi BIOS (CALL commands) can also be used irrespective of how the system was booted, with options to have the BIOS in an EEPROM (AT28C156) or load it from BASIC by loading the “msxpiext.bin” extension.

3.1. Booting with the MSXPi

When booting the MSXPi with the integrated ROM, it will try to boot into MSX-DOS 1.03 in a floppy disk image stored in the Raspberry Pi. After a successful boot, drive A: and B: will be available for the user. These drives are stored in the Raspberry Pi as .dsk disk images:

/home/pi/msxpi/disks/msxpiboot.dsk (Drive A:)

/home/pi/msxpi/disks/tools.dsk (Drive B:)

Note that booting from the integrated MSX-DOS 1 require the Raspberry Pi to complete its boot sequence and start the msxpi-server.py - for this reason, a MSX cold boot (power on) may take a few minutes to successfully boot into the MSX-DOS.

To bypass the MSX-DOS 1.03 boot and go directly to BASIC, keep ESC pressed during the boot sequence.

To boot from another interface (for example, with Nextor), insert that interface in a lower slot, and the MSXPi in the higher slot, and power on the MSX. To skip the MSXPi waiting time during the boot, keep ESC pressed during the boot sequence.

3.2. Updating the MSXPI Client and Server components

Updating using network is always a risky activity - do this with care and have also a keyboard and HDMI monitor around as last resource to recover the Raspberry Pi from blocking errors. Alternatively, a remote access via ssh is very handy, and recommended in case the update from the MSX-DOS fails.

Note: *Depending on what version you are, it might be necessary to manually update MSXPIUPD.BAT from the github repository. In case you judge its necessary, it can be updated with* *this command:*

pcopy https://github.com/costarc/MSXPi/raw/master/software/target/msxpiupd.bat

To update the MSXPi client side ("P" commands in the MSX Disk or SD CARD), simply run the command:

MSXPIUPD.BAT

The progress is displayed in the screen, and errors will be easily spotted.
Before running the command, make sure that networking is configured and working properly.

To update the MSXPi server components, the following commands can be used from the MSX-DOS:

pcd /home/pi/msxpi
prun wget https://tinyurl.com/MSXPi-Setup
prun chmod 755 MSXPi-Setup
prun ./MSXPi-Setup

These commands should only be executed after verifying that the network is working properly, because once MSXPi-Setup is executed there will be no output to the screen.
This update takes up to five minutes, and the MSX prompt will be locked until the Raspberry Pi is rebooted - at this point, the MSX returns the command prompt to the user.
Verify the MSXPi status and version with the PVER command.

3.3. Dealing with Errors and Instabilities

Because there are many parts involved in the project, and sometimes the conditions are less favourable, the MSXPi may fail. This may occur due to electric and electronic interferences and signal quality in the bus, which varies from in the several MSX models out there.
The MSXPi is very stable in MSX clones such as Zemmix, and also on MSX with more recent SoC CIs, but may be more unstable in older MSX models.

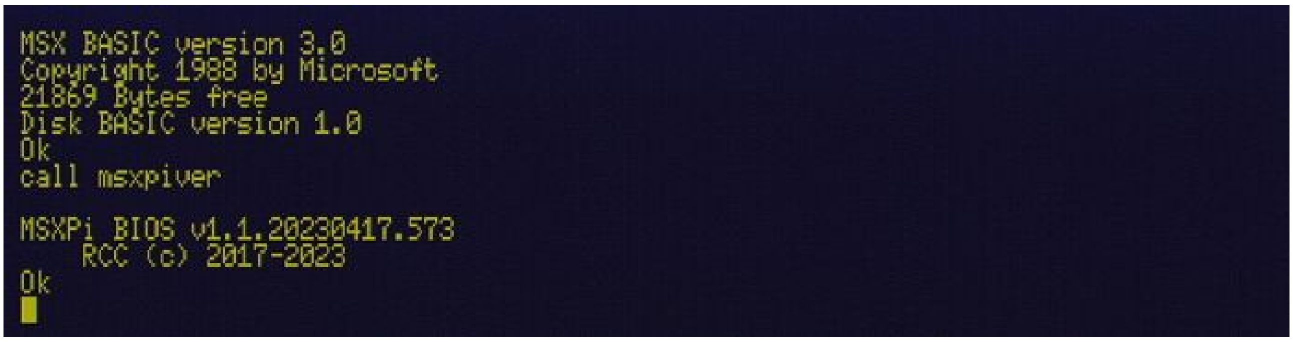
Even though there is error detection and retries in the basic transfer routines (so the developer don't need to worry about detecting errors in their code), sometimes the transfer just fail and the MSXPi enters in recovery mode.

Use ESC: **The user may interrupt an ongoing operation or recover the MSX prompt if it is apparently hanging by pressing ESC key, which will also cause the MSXPi to enter in recovery mode. The recovery mode takes 5 seconds - after pressing ESC, wait at least five seconds then try the MSXPi command again.**

When in recovery mode, the MSXPi will stop the ongoing transfer in the server side, and re-enter in "command" mode, that is, will abandon the previous operation and start listening for commands. This may take up to five seconds, and the user should not immediately retry any command, giving time for the MSXPi to recover properly.

When booting from the integrated MSX-DOS, things might get more complicated, because the control is with the MSX-DOS itself, which reads sectors and try to load the msxdos.sys and command.com during the boot. If there is an error during this phase and even after the retries the MSX-DOS cannot get the correct sectors to be transferred, the boot sequence is compromised with some unpredictable results, although most of the times the MSX will simply continue the boot sequence and boot from another connected interface, or jump to BASIC.

From BASIC, the user can verify if the MSXPi BIOS is available by running "CALL MSXPIVER".

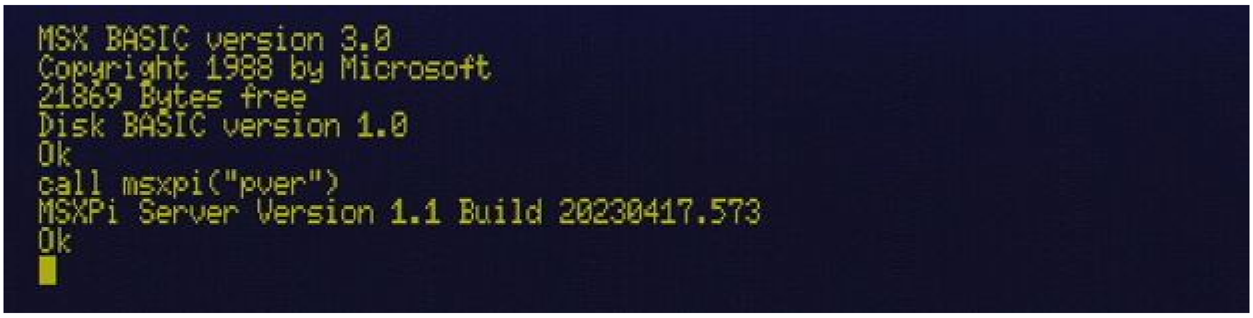


```
MSX BASIC version 3.0
Copyright 1988 by Microsoft
21869 Bytes free
Disk BASIC version 1.0
Ok
call msxpiver

MSXPi BIOS v1.1.20230417.573
    RCC (c) 2017-2023
Ok
```

When the BIOS is available, some commands can be used to verify if the MSXPi server if operation:

CALL MSXPI("pver")



```
MSX BASIC version 3.0
Copyright 1988 by Microsoft
21869 Bytes free
Disk BASIC version 1.0
Ok
call msxpi("pver")
MSXPi Server Version 1.1 Build 20230417.573
Ok
```

If the command stop responding or return "Connection Error", use ESC if necessary to interrupt the command, wait 5 seconds and try again. After retrying and it still does not work, the Raspberry Pi might need to be inspected, which requires login in and repairing potentials issues. The next section discuss some potential work around and advice to deal with some issues - if none of these work, or if the user is not technically able to perform them, the next best option is to re-write the SD card for the MSXPi.

3.3.1 Failed MSX-DOS Boot with Operational MSXPi

MSXPi failed to boot from the integrated MSXPi disk image, however after running the tests in the previous section, it was determined that the MSXPi is operational – this might imply that the msxpiboot.dsk disk image is corrupted, and needs to be replaced. Before replacing the boot disk image, the following tests may be done as an attempted:

- Try again to boot into the MSX-DOS – and the MSX has a RESET button

  If the MSX has a reset button, use it. This will boot the MSX, but will not cut the power for the Raspberry Pi, which means it will keep running the MSXPi server and the MSX may be able to boot from the disk image if it was not corrupted.

- Boot into MSX-DOS - MSX do no have RESET button

  From BASIC, enter these two lines of code:

  defusr0 = 0

  a = user(0)

- Stay in BASIC and run BASIC programs that use MSXPi

  Try the command "*files*" to check if the DOS system is responding.

  Try also "*POKE &HF346,1*" then "*files*".

  And finally, try "*POKE -1,170*" followed by "*files*".

  If none of these work, you may have to switch off the MSX (which will power off also the Raspberry Pi), and ON again and see if the MSX boots from the MSXPi DOS disk image.

To restore the msxpiboot.dsk image, simply replace /home/pi/msxpi/disks/msxpiboot.dsk by the most recent version in the github repository. This is automatically done when running the updater batch (msxpiupd.bat), which can be used booting from another disk drive interface.

3.3.2 Failed MSX-DOS Boot with Failing MSXPi

In this situation, the user can either switch off/on the MSX and hope for the MSXPi to boot properly, or use a computer with a ssh client (cygwin or putty in Windows) to login to the Raspberry Pi, run the MSXPi server manually and troubleshoot by inspecting the messages in the console.

If MSXPi respond to commands at some point, but refuses to boot from the MSX-DOS disk image, it could be that the disk image was corrupted due to repetitive power off. In this case, copy a fresh msxpiboot.dsk to the Raspberry Pi to see if it solve the problem - this can be done downloading the fresh disk image from the MSXPi git repository and copying to the Raspberry Pi.

From a Linux, Mac or Cygwin shell (under Windows) using scp command:

scp msxpiboot.dsk pi@raspberrypi:msxpi/disks/

Remember that the Raspberry Pi Zero is powered by the MSX. When the MSX is powered off, so is the Raspberry Pi. Ideally, the user should always run "pshut" from DOS, or call msxpi("pshut") from BASIC before powering off the MSX, to reduce the chances of having the SD Card corrupted.

3.4. Using the MSXPi Commands

P commands are the ".com" commands available under the DOS system (MSX-DOS1/2,Nextor). These commands works tin the same way independent of the MSX-DOS version, and are described in details in this section because they are the "core" MSXPi commands.

3.4.1 AT28C256

Write a ROM file to the EEPROM. The ROM may be 8KB, 16KB or 32KB.

It's possible to write two ROM files (8KB or 16KB each) and switch between them by switching the jumper A14/A15 - the ROMs must be previously merged, each ROM stored in the beginning of 16KB area.

Examples:

at28c256 /i (Display headers of all identified ROMs)

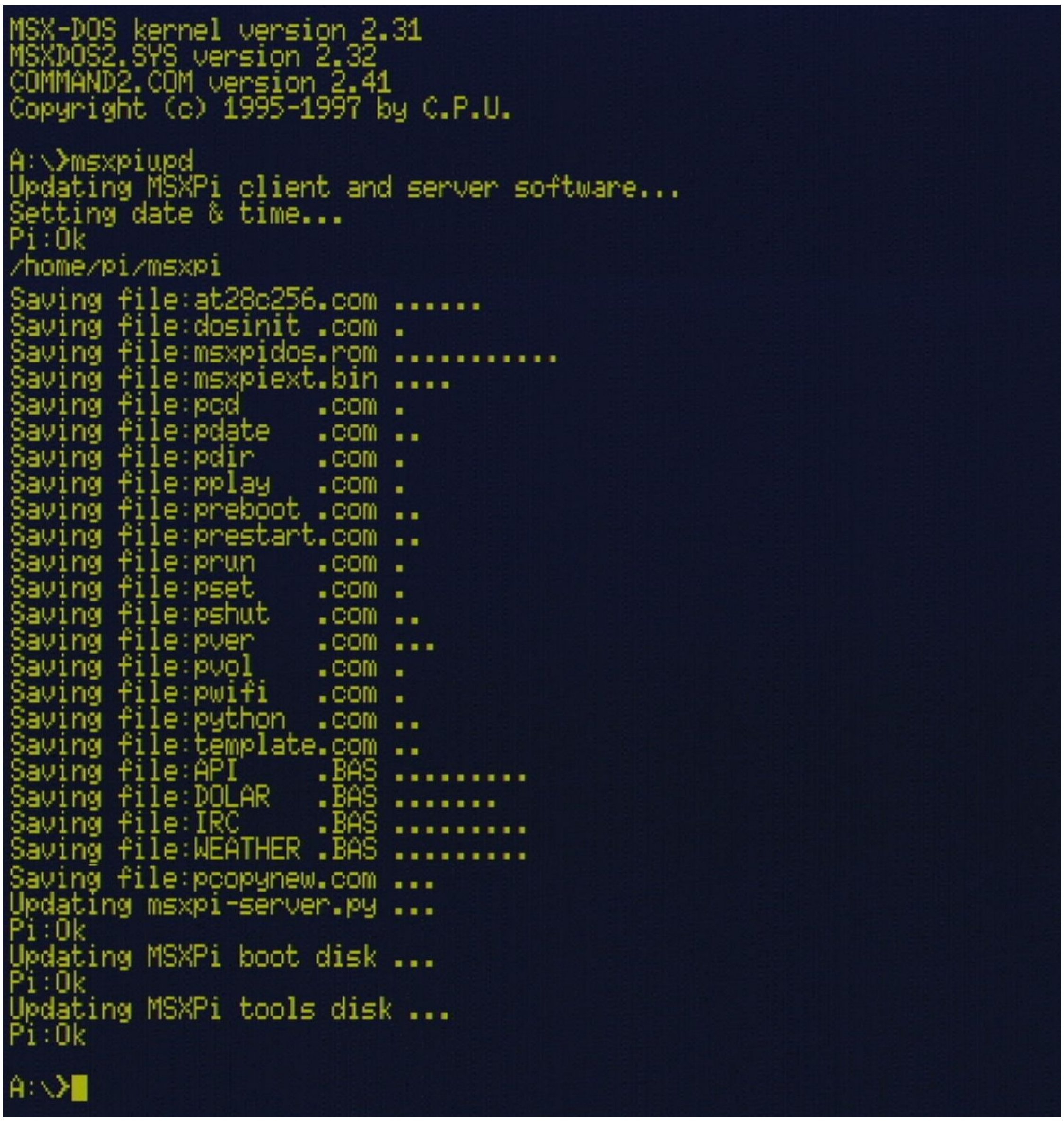
at28c256 /s 2 MSXPIDOS.ROM (write MSXPi ROM to MSXPi connected msxpiupdn slot 2)

3.4.2 MSXPIUPD

Perform a full upgrade of the MSXPi client and server software - the ROM is downloaded but not written to the EEPROM, which needs to be done manually using at28c256.com programmer.

This command download all P commands from github to the MSX drive - existing versions re overwritten. The server components are also updated, including the two disk images used by the MSX-DOS 1.03.

In the MSX-DOS 2, the output is nicely formatted as show in next picture. In the MSX-DOS 1, there will be some error messages every time the “ECHO” command is used in the batch file, but the files will be updated successfully.



```
MSX-DOS kernel version 2.31
MSXDOS2.SYS version 2.32
COMMAND2.COM version 2.41
Copyright (c) 1995-1997 by C.P.U.

A:\>msxpiupd
Updating MSXPi client and server software...
Setting date & time...
Pi:Ok
/home/pi/msxpi
Saving file:at28c256.com ......
Saving file:dosinit .com .
Saving file:msxpidos.rom ...........
Saving file:msxpiext.bin ....
Saving file:pcd     .com .
Saving file:pdate   .com ..
Saving file:pdir    .com .
Saving file:pplay   .com .
Saving file:preboot .com ..
Saving file:prestart.com ..
Saving file:prun    .com .
Saving file:pset    .com .
Saving file:pshut   .com ..
Saving file:pver    .com ...
Saving file:pvol    .com .
Saving file:pwifi   .com .
Saving file:python  .com ..
Saving file:template.com ..
Saving file:API     .BAS .........
Saving file:DOLAR   .BAS .......
Saving file:IRC     .BAS .........
Saving file:WEATHER .BAS .........
Saving file:pcopynew.com ...
Updating msxpi-server.py ...
Pi:Ok
Updating MSXPi boot disk ...
Pi:Ok
Updating MSXPi tools disk ...
Pi:Ok

A:\>
```

3.4.3 PDATE

Sets the MSX date and time by reading these information from Raspberry Pi. This command does not accept any parameter.

3.4.4 PCD

Sets the path for the MSXPi commands pdir and pcopy. Is called without parameters, will display the current path.

The path may be a Raspberry Pi filesystem path, a http/ftp/smb url, or one of the three virtual devices below. Any of these three devices can be changed via ***pset*** command.

- **m** A user local network resources. The default path for m: is ftp://192.168.1.100
- **r1** A internet location. Default path is the msxarchive.nl roms for MSX1

- **r2** A internet location. Default path is the msxarchive.nl roms for MSX2

Examples:

pcd

pcd /home/pi

pcd m:

pcd r1:

pcd http://www.msxarchive.nl/pub/msx/

3.4.5 PDIR

Show contents of current path in the MSXPi when no path is passed as parameter.

Show contents of given path when passed as parameter.

Note that the path can be local to the Raspberry Pi and also a remote / network resource.

Examples:

pdir /home/pi

pdir http://www.msxarchive.nl/pub/msx

3.4.6 PCOPY

Copy a file from a Raspberry Pi path (filesystem or network) to the MSX drive.

The file can be in the Raspberry Pi s card (any folder), in the network (ftp/http/smb) on in one of the three virtual devices (m, r1 and r2).

pcopy accepts the parameter "/z" to decompress a file, lookup its original name inside the compressed archive, and use it to save in the MSX drive.

Examples:

pcopy /home/pi/msxpi/msxpi.ini msxpi.ini

pcopy ftp://192.168.1.100/pver.com

pcopy m:pver.com

pcopy /z r1:frogger.zip (download frogger.zip from msxarchive, unzip, save as frogger.rom)

3.4.7 PPLAY

Play music and audio from RPi SD card (soon also to play multimedia over the network). PPLAY can play a music file, pause, resume and stop it. The audio can also played in a loop. Once the music is started, a unique ID is returned to the MSX, which can be used with the pause/resume/stop commands.

- Options implemented so far:
  - play (start playing a multimedia file or stream)
  - loop (play the same audio until a stop is sent to RPi)
  - pause (pause the audio)
  - resume (restart the audio previously pause)
  - stop (stop the audio)
  - getids (list the unique ID of the audio playing)
  - getlids (list the unique ID of the looping audio)

Examples:

pplay play Aleste.mp3

1465

*pplay pause **1465***

*pplay resume **1465***

*pplay stop **1465***

***Note:** 1465 aboce is the unique ID of the audio playing. It will be different each time a new audio is played.*

pplay loop Aleste.mp3

pplay getlids

pplay getids

3.4.8 PRUN

Run a command in the Raspberry Pi. Note that commands that require inputs are not supported.

Pipe is supported, but must be replaced by "::" in the MSXPi.

Examples:

prun ls -l /etc

prun cat /home/pi/msxpi/msxpi.ini

prun ps -ef :: grep msx

prun wget http://www.msxarchive.nl/pub/msx/games/roms/msx1/frogger.zip

3.4.9 PSET

This command is used to manage the MSXPi variables, such as the WiFi ssid, WiFi password, virtual drives, MSXPi current path, disk images and other user variables.

The variables are saved in a file: /home/pi/msxpi/msxpi.ini and loaded when MSXPi starts.

When used to set the disk images for MSXPi, this command will enforce the loading of the new disk image, making it available immediately.

***Note**: It's recommended to not change variable DriveA, because it defines the disk with MSX-DOS and the P commands for the MSX-DOS 1.03 boot. Changing this variable may lock the user out of the DOS, requiring manual changes to the msxpi.ini file to recover the boot.*

Examples:

pset

pset WIFISSID My Wifi Name

pset WIFIPWD MyWifiPass

pset DRIVE1 /home/pi/msxpi/disks/diskb.dsk

3.4.10 PWIFI

Display the current network interface information, and set the WiFi using the Wifi variables

Examples:

pwifi

pwifi set

*Note: "**pwifi set**" will use "**WIFISSIDW**" and "**WIFIPWD**" MSXPi variables to configure the Wifi interface. After "pwifi set" command, a reboot is required for the Raspberry Pi to enable and acquire the new network configuration.*

3.4.11 PRESTART

Restart the MSXPi Server (msxpi.server.py) in the Raspberry Pi.

3.4.12 PREBOOT

Reboot the Raspberry Pi.

3.4.13 PSHUT

Shutdown the Raspberry Pi. This command should be used before powering off the computer.

4 Developers Guide

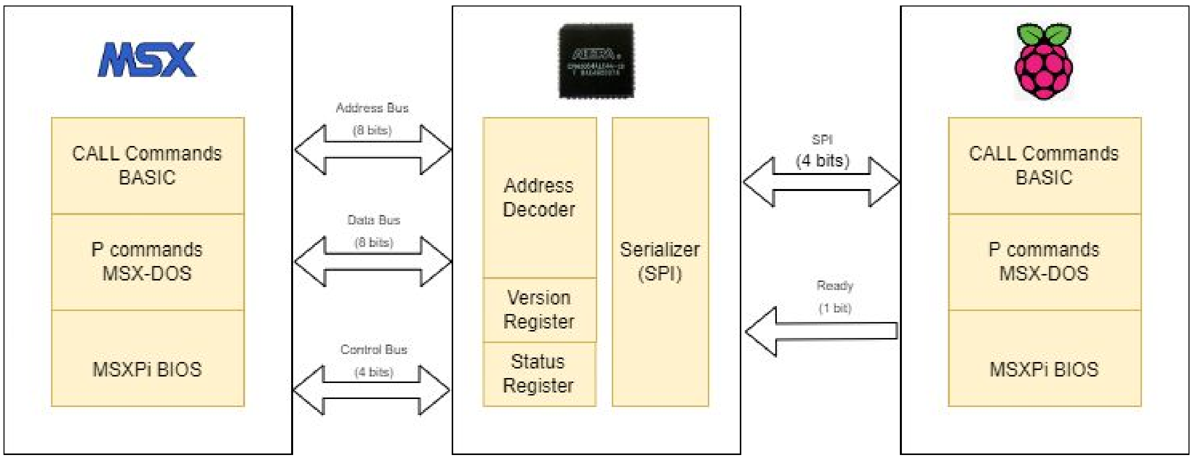
This section is directed to developers that want to create new MSXPi programs, in either BASIC or Assembly in the MSX side. Note that all code on the server side is always Python.

4.1. MSXPi Architecture

MSXPi contains three main components:

- MSX client software for both MSX-DOS and BASIC
- MSXPi interface (cartridge to insert in the MSX slot)
- Raspberry Pi server software

These components are connected and interact as shown in this architecture diagram:



The interface is enabled when the MSX write or read the IO ports 0x56 and 0x57. These ports are decoded by the CPLD in the interface, serialize the data in the data bus and transfer to the Raspberry Pi, which convert the data back bytes and process it as defined in the MSXPi protocol (see next section).

4.2. MSXPi Protocol

The recommended method to develop for MSXPi is to use the BIOS and routines available. These routines implements all the low level aspects of the communication, making it easier to implement new commands and exchange data with the Raspberry Pi.

All programs must always start with a command being sent to Raspberry Pi. This command has a fixed length of 8 valids characters in the range A-Z or a-z (case is actually irrelevant). The command is always parsed and processed by the msxpi-server.py program, which will then call a function in with the same same within the program. This function must have been implemented by the user otherwise the server will return an error.

Data is transferred in blocks of 8, 128, 256 or 512 bytes, depending on the stage and function being implemented. Its possible to use blocks of any size when calling the basic transfer routines directly (SENDDATA and RECVDATA).

Each block contains also three (3) additional bytes used for header, which must be considered when reserving buffer areas for the commands. This header contains the return code provide by the Raspberry Pi, and the size of useful data in the block (this will be covered in details in the development sections below).

Once the function is called, it’s up to the programmer to implement the required behaviour, which may include passing additional parameters, sending and receiving data on both directions, etc. The MSXPi BIOS routines have a number of functions the developer can use to implement their programs, which will be described later.

Most of the MSXPi commands follow this flow:

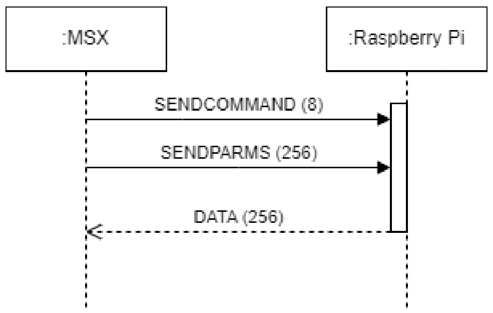
1. Send a command (8 bytes)
2. Send the parameters (256 bytes)
3. Receive the response (256 bytes or more)

The command name (8 bytes string) corresponds to a function in the msxpi-server.py running in the Raspberry Pi. These commands (such as “pcd”) will be executed in the Raspberry Pi, and return the output as a 256 bytes data block – in some cases, the returned data is larger than 256 bytes, in which case the P command must be prepared to detect and process correctly the additional blocks – failing to do this will cause the MSXPi to enter an “out of sync” state and fail.

There are many variations of the high level protocol, but as previously mentioned, they all must start with a command being set to Raspberry Pi. Below it’s shown the main possible variations of the structure and sequence of actions in MSXPi commands.

Send a command, followed by the parameters. Expect to to receive at least one block of data back as response. This is the most common structure, because the response will contain useful data to be presented to the user in either successful or failure situations:

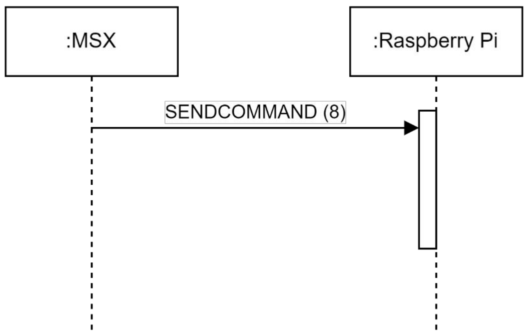
- Output of the command executed in the Raspberry Pi
- Error message when the command failed



MSXPi Protocol 1

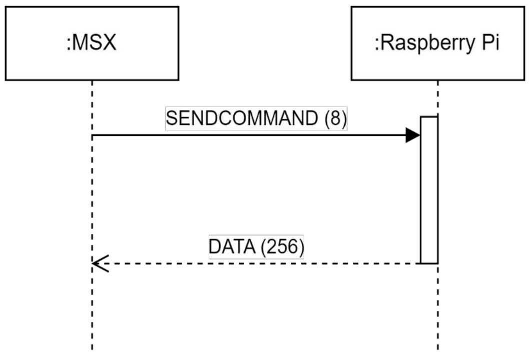
This structure is used in most of the P commands, and is also implemented in the CALL MSXPi for BASIC development. The user can easily implement additional functionality for MSXPi using this structure by cloning the TEMPLATE code (in the Clients/src folder) and the "template()" function in the msxpi-server.py.

Send a command, and do not expect a response:



MSXPi Protocol 2

Send a command and expect one block of data (256 bytes):

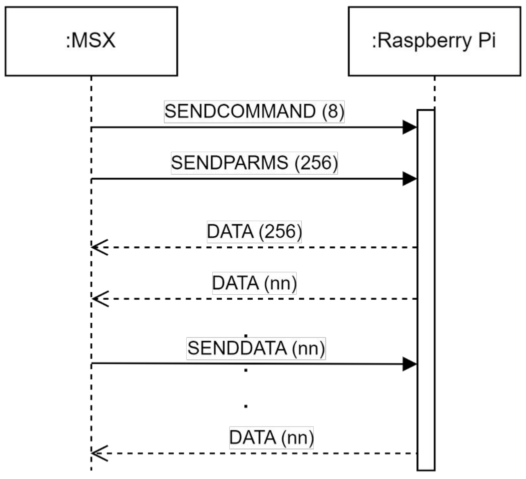


MSXPi Protocol 3

Send a command, send additional parameters, and receive many blocks of data. Optionally, send also additional data during the execution of this command.

This is a more complex implementation, and there are different ways this structure can be implemented, by either locking on not locking the msxpi-server.py. Study the command PCOPY (locking the server to a single command) and the BASIC program IRC.BAS (not locking the server).

“Locking the server” means that the command takes over the MSXPi until its completely finished. “Not locking the server” means the operations are asynchronous, and one program span across several calls to MSXPi, allowing other commands to be used in between calls.



MSXPi Protocol 4

4.3. Developing in Basic

Programs can be developed in BASIC to use MSXPi resources by using three BIOS commands:

- CALL MSXPi – Send a command and parameters to Raspberry Pi, and receive back a reply
- CALL MSXPISEND – Send a block of data to Raspberry Pi
- CALL MSXPIRECV – Receive a block of data from Raspberry Pi

All commands need a buffer address, which can be provided by the developer. If a buffer area is not provided, the MSXPi BIOS command will automatically reserve one with 259 bytes at the top of the RAM. Note that leaving the BIOS to reserve the buffer might be dangerous, because if the commands return data larger than the buffer, it will start to overwrite system ram (and the STACK) at the top of the RAM, causing a crash.
The recommended way to allocate a buffer is to do it explicitly in the command, with enough capacity for the data that is expected to be received. For example, if the developer leaves the buffer allocation responsibility for the MSXPi BIOS while using executing CALL MSXPi(“prun ls -lR /”),

the computer will crash because the amount of data returned by the command is immensely larger than the buffer, consequently causing a crash.

4.3.1 CALL MSXPi

Syntax:

CALL MSXPi(“<stdout>,<buffer address>,<command>”)

Examples:

```
call msxpi(“pdir”)
call msxpi(“0,pdir”)
call msxpi(“1,pdir /home/pi/msxpi”)
call msxpi(“2,D000,pdir /etc”)
```

Executes a command (function) 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 Raspberry Pi
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 memory starting in the address provided in the command (the buffer). May corrupt the top RAM and stack if data is larger than the available space in the buffer.

Buffer Address: This is a string composed of four hexadecimal digits for a memory area to receive data from Raspberry Pi – for example, “C000”. It must have at least 259 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. Parameters are also accepted.

CALL MSXPi uses a buffer of size BLKSIZE (259 bytes) to receive data from Raspberry Pi. Each 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)

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 in each block contain the size of the useful data in that block. The client program must control how much data and blocks is read from Raspberry Pi based on this header.

The size for the useful data in the block can be calculated in BASIC as follows (assuming it was passed "D000" in the CALL MSXPi command):

```
SIZE=PEEK(&HD001)+256*PEEK(&HD002)
```

Note that this buffer structure is hard-coded in the MSXPi BIOS commands, and need to be respected by the MSXPi Server commands running in the Raspberry Pi.

The return codes in each block can be one of:

RC_TXERROR	Connection error or checksum error after all retries. This code is generated by the MSX client software because the transfer failed. There is not valid data in the buffer.
RC_SUCCESS	Operation successful. This code is returned by the Raspberry Pi. There is no blocks to be transferred. Buffer contain valid data.
RC_FAILED	Operation failed. This code is returned by Raspberry Pi. There is no more blocks to be transferred. Buffer may contain a valid error message.
RC_READY	Operation successful. This code is returned by the Raspberry Pi. There are more blocks to be transferred. Buffer contain valid data.

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: Raspberry Pi did not receive the command or it failed the checksum. The return code will be RC_TXERROR (set by the MAX due to lack of communication with Raspberry Pi)

2. Raspberry Pi received the command, processed it but it failed: The return code will be set by Raspberry Pi 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. Raspberry Pi received the command, processed and it succeed. The return code is RC_SUCCESS, and there will be data available in the buffer.

4. Raspberry Pi received the command, processed and it succeed. The return code is RC_READY, meaning there is another blocks of data ready to be transferred. When receiving this return code, the client software must read another block, otherwise the service will be stuck waiting to send the data, causing an "out of sync" situation.

4.3.2 CALL MSXPISEND

"buffer_address" is a four digit hexadecimal number.

Send contents of buffer to Raspberry Pi (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.

The following example will send three ASCII characters to Raspberry Pi. Because the size of a data block is always 256 bytes (plus header), the string is terminated with zero to let the server side program to know when the string ends.

```
10 GOSUB 100 ' Send a command to Raspberry Pi
20 RC = PEEK(&HD000): ? HEX$(RC) ' Get Return Code
50 END
99 ' Send data to Raspberry Pi – a text command in this case
100 POKE &HD003,ASC('T') ‘ Data should be stored from Buffer +3 to skip the header area
110 POKE &HD004,ASC('S')
120 POKE &HD005,ASC('T')
130 POKE &HD006,0
140 CALL MSXPISEND("D000") ' Send the data to Raspberry Pi
150 RETURN
```

4.3.3 CALL MSXPIRECV

"buffer_address" is a four digit hexadecimal number.

Read data from Raspberry Pi and store in the buffer (it uses the RECVDATA 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.

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

```
10 GOSUB 100 ' Receive data from Raspberry Pi
20 RC = PEEK(&HD000): ? HEX$(RC) ' Get Return Code
30 IF RC = &HE0 THEN GOSUB 200 ‘ Print the data received from Raspberry Pi (assuming it is
text)
50 END
99 ' Receive some data from Raspberry Pi
100 CALL MSXPIRECV(“D000”) ' Received data from Raspberry Pi
110 RETURN
200 S=PEEK(&HD001)+256*PEEK(&HD002) ' Get size of useful data in buffer
210 FOR M = 0 TO S-1 ' Will read all bytes in the buffer
220 PRINT CHR$(PEEK(&HD003+M)); ' and print on screen.
230 NEXT M:RETURN
```

4.4. Developing in Assembly

This section describes the high level routines that can be used to support the development of new applications for the Raspberry Pi to run under MSX-DOS or BASIC.

It’s described only the recommended routines, although may others are available and can be used by the developer.

The template below show the minimum structure a MSXPi program should have, and the API libraries it must import.

```
ORG  $100

; User code – start here

; …

: ...

; User code – ends here



; Core MSXPi APIs / BIOS routines

INCLUDE "include.asm"

INCLUDE "putchar-clients.asm"

INCLUDE "msxpi_bios.asm"

buf:  equ     $

       ds      BLKSIZE
```

For a full template of a MSXPi program running under MSX-DOS, refer to "*msxpi/software/Client/src/template.asm*"

Before starting writing the program, make sure you are familiar with the MSXPi Protocol in section 4.2.

4.4.1 Common Routines

These are the most common routines needed to develop a .com program for MSXPi. They all hide the more complex lower layer of the protocol, facilitating and speeding the development.

4.4.1.1 SENDCOMMAND

Sends a command with a maximum of eight bytes to the server. The command should be defined somewhere, terminated in zero and be valid ascii characters. Register DE must contain the address of the command before calling SENDCOMMAND.

```
   ld      de,command
   call    SENDCOMMAND
           ...
           ...
   ret
command: db "pdir",0
```

4.4.1.2 SENDPARMS

This command is only valid for MSX-DOS programs, as it reads the buffer area used to store parameters passed when running .com commands (address $80). If the program does not need parameters, this routine may not be used.

The routine uses the BUF area, which should be cleared before calling SENDPARMS.

```
   ld      de,command
   call    SENDCOMMAND
  ; Clear the buffer area, which has 256 Byes (BLKSIZE)
   ld      de,buf
   ld      bc,BLKSIZE
   call    CLEARBUF
; Check if there are parameters in the command and send to Raspberry Pi
   call    SENDPARMS
```

```
; User program here
    ...
    ...
    ret
command: db "pdir",0
```

4.4.1.3 SENDDATA

Sends a block of data to the server - its the counterpart of RECVDATA. The size of the block to transfer is passed in BC and the RAM address to read the data from in DE register. Any error in the transfer will return flag C set, otherwise NC will be set.

After finishing the transfer, DE returns updated pointing to the next available address in the RAM buffer.

```
    ld    de,buf
    ld    bc,BLKSIZE
    call    SENDDATA
    jr    c,error
    ret
```

4.4.1.4 RECVDATA

Receives a block of data from the server - its the counterpart of SENDDATA. The size of the block to transfer is passed in BC, and the RAM address to store in DE register. Any error in the transfer will return flag C set, otherwise NC will be set.

After finishing the transfer, DE returns updated pointing to the next available address in the RAM buffer.

```
    ld    de,$4000
    ld    bc,BLKSIZE
    call    RECVDATA
    jr    c,error
    ret
buf: equ $
```

4.4.1.5 PRINTPISTDOUT

Prints a string to screen.

String address is passed in DE, and size in BC.

Will terminate and return if finds zero even if BC is not zero.

4.4.1.6 PRINTNLINE

Prints a ASCII 13 followed by a ASCII 10 to the screen, forcing the cursor to start of new line.

4.4.1.7 PRINT

Prints a text terminated in zero to screen.

String is passed in HL.

4.4.1.8 CLEARBUF

Write zeroes to a buffer area.

Address is passed in DE and size in BC.

4.4.1.9 Other Functions

Refer to “/software/asm-common/include/msxpi_bios.asm” for other functions and routines that can be used to accelerate development of programs in assembly for MSXPi.

5 Appendices

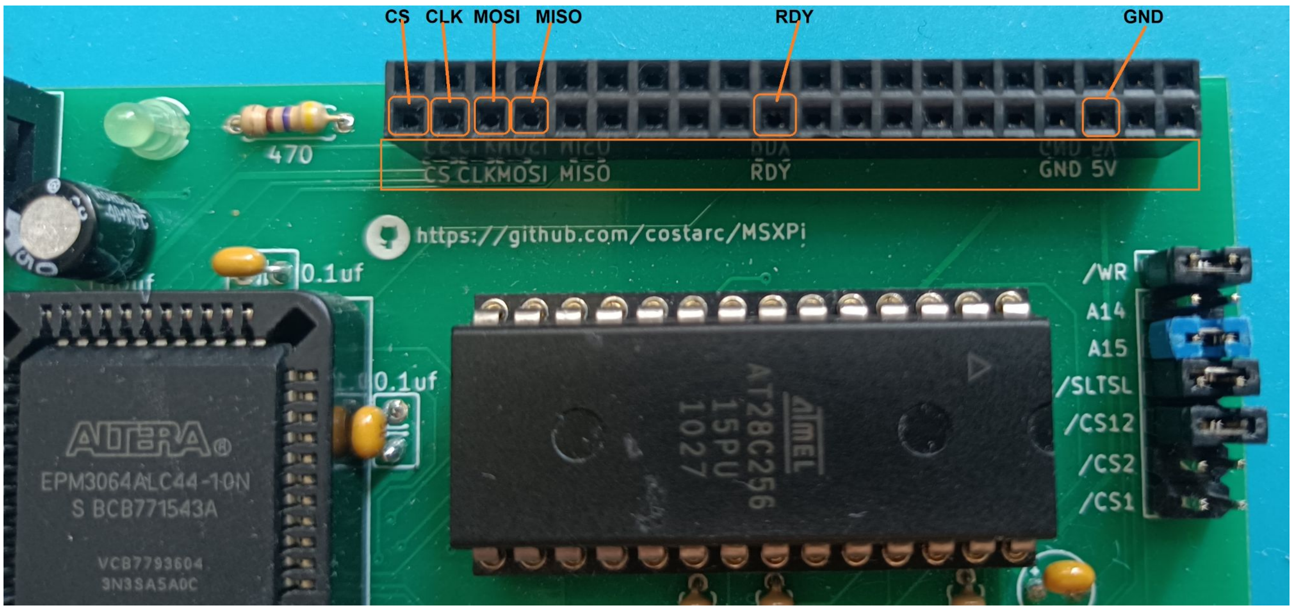
5.1. GPIO Pin Assignments

In case the user intends to connect the MSXPi to a Raspberry Pi 2/3/4, the following pinout should be used as reference. These GPIO pins are the default values as wired by default, but the interface v1.1 have optional jumpers for a different set of GPIOs, which when used, must be defined in the server via the PSET command.

Note that a connection using wires may be unstable and prone to connection failures.

MSXPi Signal	Raspberry Pi GPIO	Raspberry Pi GPIO (Optional in PCB v1.1 Rev0)	Raspberry Pi Header Pin (Primary / Optional)
CS	21	8	40 / 24
CLK	20	11	38 / 23
MOSI	16	10	36 / 19
MISO	12	9	32 / 21
SPI_READY	25	24	22 / 18
GND	GND	GND	6/9/14/20/25/30/34/39

/



5.2. Appendix 2: Pi - GPIO Pin Numbering

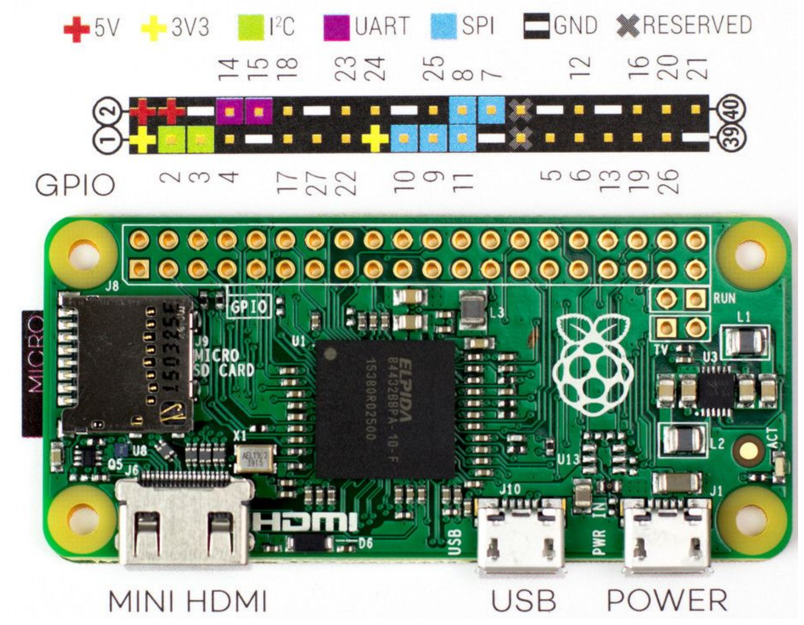


Illustration 4: GPIO models B +, 2, 3 and Zero.

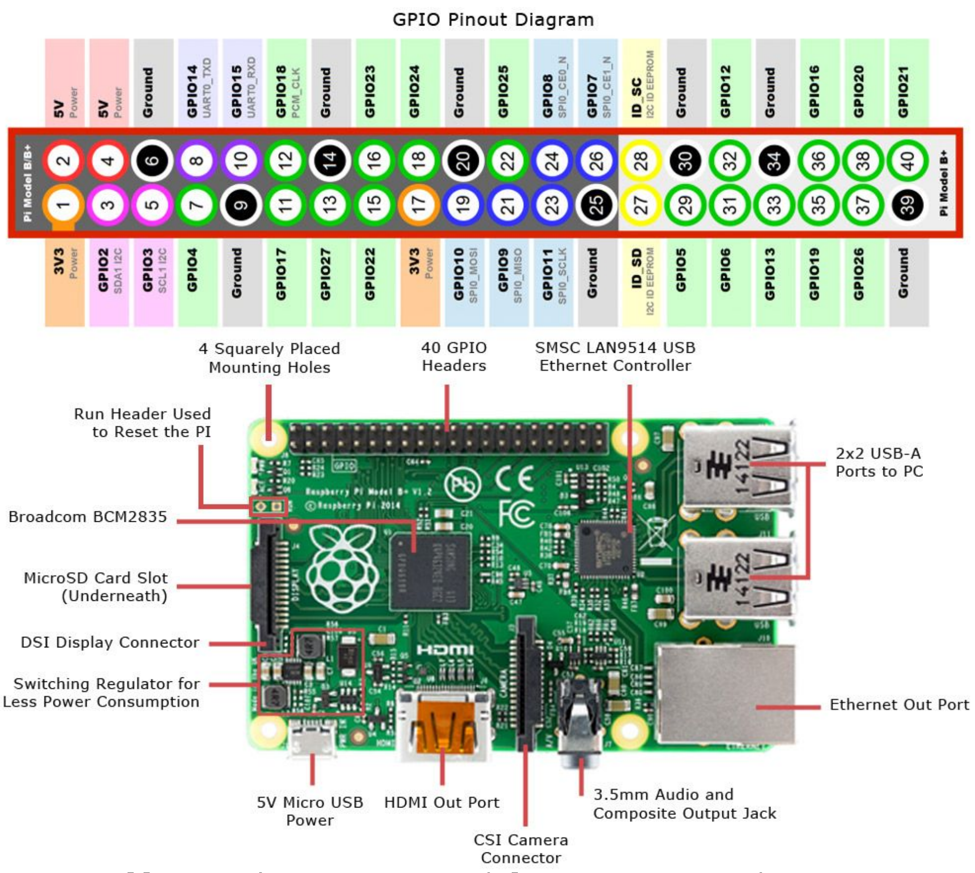


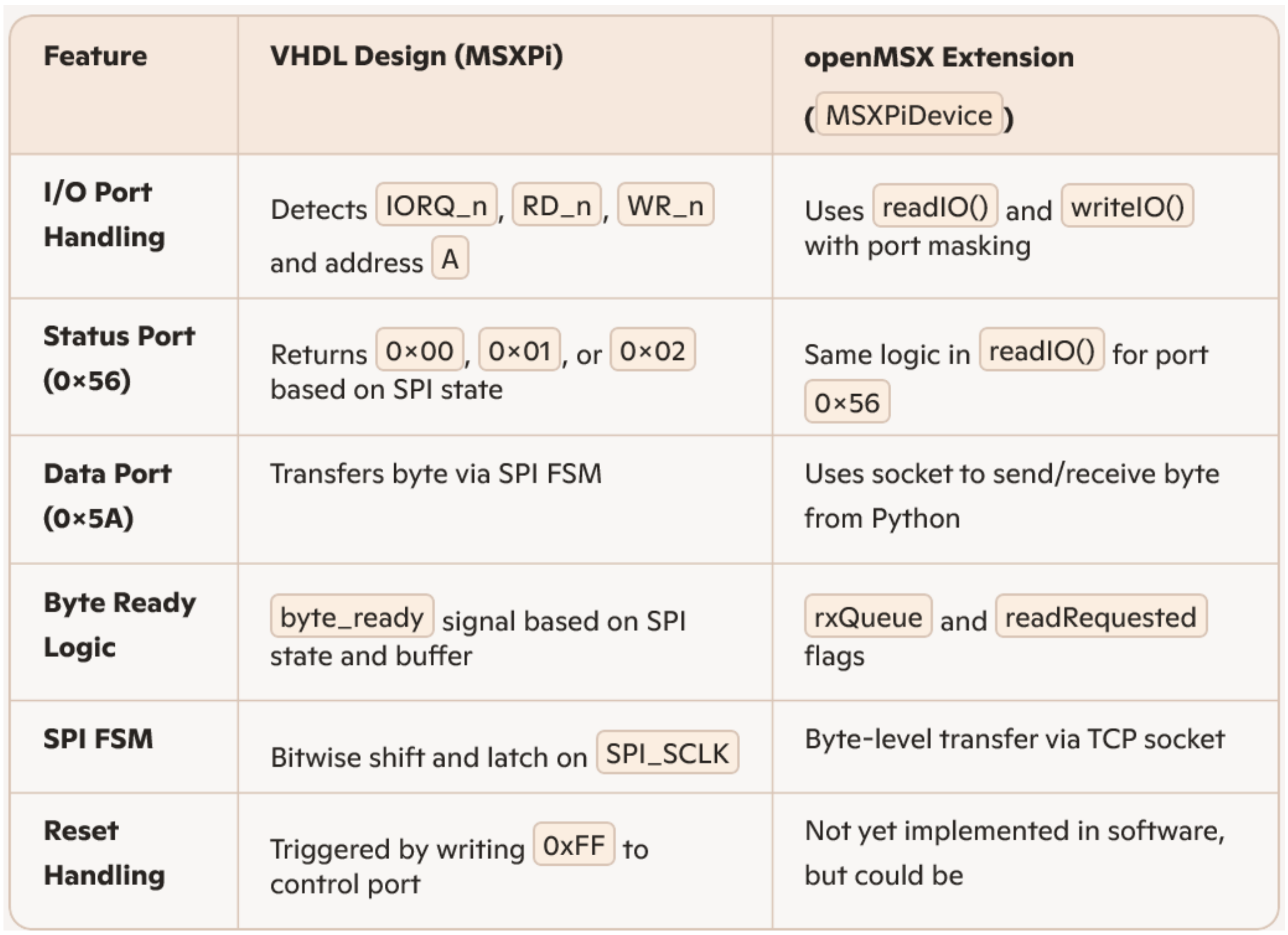
Illustration 5: GPIO models B +, 2, 3 and Zero.

5.3. Appendix 3: MSXPi Extension for openMSX

Overview

This section compares the VHDL implementation of the MSXPi interface with its software counterpart in the openMSX emulator. Both designs aim to bridge MSX with a server-side component (usually running in Python) hosted either in the Raspberry Pi (in the physical interface) on in the host where openMSX is running.

Functional Comparison



Feature	VHDL Design (MSXPi)	openMSX Extension (`MSXPiDevice`)
I/O Port Handling	Detects `IORQ_n`, `RD_n`, `WR_n` and address `A`	Uses `readIO()` and `writeIO()` with port masking
Status Port (0×56)	Returns `0×00`, `0×01`, or `0×02` based on SPI state	Same logic in `readIO()` for port `0×56`
Data Port (0×5A)	Transfers byte via SPI FSM	Uses socket to send/receive byte from Python
Byte Ready Logic	`byte_ready` signal based on SPI state and buffer	`rxQueue` and `readRequested` flags
SPI FSM	Bitwise shift and latch on `SPI_SCLK`	Byte-level transfer via TCP socket
Reset Handling	Triggered by writing `0xFF` to control port	Not yet implemented in software, but could be

Key Differences

- **Timing Model**:
  VHDL uses `rising_edge(SPI_SCLK)` for precise bit-level timing. The openMSX extension abstracts this using byte-level socket communication, which is asynchronous and buffered.
- **Concurrency**:
  VHDL is inherently parallel, with signals reacting to clock edges. The software version uses threads and queues to simulate concurrency and data flow.

- **Bus Direction and Tristate Logic**:
  The VHDL design drives `BUSDIR_n` and tristates the `D` bus. The openMSX extension simulates read/write behaviour but does not model electrical contention.
- **SPI Signal Handling**:
  VHDL directly manipulates `SPI_MOSI`, `SPI_MISO`, and `SPI_CS`. The software version emulates SPI indirectly via a TCP protocol, abstracting the physical layer.

What This Means

The VHDL and openMSX implementations are **functionally equivalent** in terms of protocol, port usage, and control flow. They differ in execution model — hardware vs software — but both achieve the same goal: enabling MSX to communicate with a server-side application implemented in Python using a clean, byte-oriented SPI protocol.

This equivalence allows developers to:

- Test and debug MSXPi software using openMSX before deploying to hardware
- Maintain consistent behaviour across emulated and physical environments
- Extend or refine the protocol in either domain with confidence