



MSXPi Interface Version 1.1

User and Developers Guide

Table of Contents

1 Introduction.....	3
1.1. Whats is New.....	3
2 Getting Started.....	5
2.1. Assembling the Interface.....	5
2.2. Preparig the SD Card for Raspberry Pi.....	5
2.2.1 Using the MSXPi Pre-Installed SD Card.....	5
2.2.2 Installing and configuring Raspbian from Scratch.....	5
2.3. Preparing the MSX.....	5
3 Users Guide.....	6
3.1. Booting with the MSXPi.....	6
3.2. Dealing with Errors and Instabilities.....	6
3.2.1 Failed MSX-DOS Boot with Operational MSXPi.....	7
3.2.2 Failed MSX-DOS Boot with Failing MSXPi.....	8
3.3. Using the MSXPi Commands.....	8
3.3.1 at28c256.....	8
3.3.2 msxpiupd.....	9
3.3.3 pdate.....	9
3.3.4 pcd.....	9
3.3.5 pdir.....	10
3.3.6 pcopy.....	10
3.3.7 prun.....	10
3.3.8 pset.....	11
3.3.9 pwifi.....	11
3.3.10 prestart.....	11
3.3.11 preboot.....	11
3.3.12 pshut.....	12
4 Developers Guide.....	13
4.1. MSXPi Architecture.....	13
4.2. MSXPi Protocol.....	13
4.3. Developing in Basic.....	16
4.3.1 CALL MSXPI.....	16
4.3.2 CALL MSXPISEND.....	19
4.3.3 CALL MSXPIRECV.....	19
4.4. Developing in Assembly -.....	20
4.4.1 Common Routines.....	20
4.4.1.1 SENDCOMMAND.....	20
4.4.1.2 SENDPARMS.....	20
4.4.1.3 SENDDATA.....	20
4.4.1.4 RECVDATA.....	20
4.4.1.5 PRINTPISTDOUT.....	20
4.4.1.6 PRINTNLINE.....	21
4.4.1.7 PRINT.....	21
4.4.1.8 CLEARBUF.....	21

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 interface that can be used with most 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 ROM. 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 ROM also contain the MSXPi BIOS with a set of CALL commands that can be used BASIC.
- MSX-DOS commands - these are "P" .com commands that implement a series of functionalities on MSX-DOS allowing greater experience with MSXPi, such as setting system clock and access to internet resources.
- MSXPIEXT.BIN is the MSXPi BIOS implementaton 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 the 4000h cartridge area, becoming available for BASIC programs.

1.1. Whats is New

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 straightforward 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 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

2.1. Assembling the Interface

2.2. Preparig the SD Card for Raspberry Pi

2.2.1 Using the MSXPi Pre-Installed SD Card

2.2.2 Installing and configuring Raspbian from Scratch

2.3. Preparing the MSX

3 Users Guide

MSXPi can be used as a standalone 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 storage and along with all the features the MSXPi provides.

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 running 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 minnutes to sucessfully 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. Dealing with Errors and Instabilities

Because there are many parts involved in the proejct, and sometimes the conditions are less favorable, 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 fbe more unstable in older MSX models.

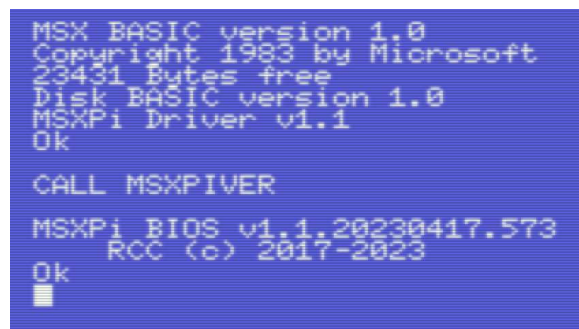
Even though there is error detection and retries in the basic transfer routines (so the developer don't need to worry about detecting erorrs in their code), sometimes the transfer just fail and the MSXPi enters in recovery mode.

The user may interrupt an ongoing operation by pressing ESC, which will also cause the MSXPi to enter in recovery.

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 connector interface, or jump to BASIC.

From BASIC, the user can verify if the MSXPi BIOS is available by inspecting the boot messages and looking for "MSXPi Driver v1.1", and also running "CALL MSXPIVER".



```
MSX BASIC version 1.0
Copyright 1983 by Microsoft
23431 Bytes free
Disk BASIC version 1.0
MSXPi Driver v1.1
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")

If the command hangs out, stop responding or return "Connection Error", wait 5 seconds and try again. After retrying, if it still does not work then the the Raspberry Pi might need to be inspected, which requires logging in and repairing potentials issues. Next is given some advice on potential work arounds for 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.

3.2.1 Failed MSX-DOS Boot with Operational MSXPi

MSXPi failed to boot from the integrated MSXPi disk image, however after runnign the tests in the previous section, it was determined that the MSXPi is operational. The next steps depends on what the user wants to do.

- Boot into the MSX-DOS - MSX has 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 not have Reset button

From BASIC, enter these two lines of code:

```
defusr(0) = 0
```

```
a = user(0)
```

- 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.

3.2.2 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.

3.3. Using the MSXPi Commands

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

3.3.1 at28c256

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

It's possible to write two ROMs (16KB each) and switch between them by switching the jumper A14/A15 (the ROMs must be previously merged).

Examples:

at28c256 /i (Display headers of all identified ROMs)

at28c256 /s 2 MSXPIDOS.ROM (write MSXPi BIOS to the EEPROM in slot 2)

3.3.2 msxpiupd

Perform a full upgrade of the MSXPi (client and server - the ROM is not updated).

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.

3.3.3 pdate

Sets the MSX-data and time by readingn these informaton from Raspberry Pi. This command does not accept any parameter.

3.3.4 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/

3.3.5 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
```

3.3.6 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 fo the three virtual devices (m, r1 and r2).

pcopy accepts the parameter "/z" to desompress 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)
```

3.3.7 prun

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

Pipe is suported, 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
```

3.3.8 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

3.3.9 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.

3.3.10 prestart

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

3.3.11 preboot

Reboot the Raspberry Pi.

3.3.12 pshut

Shtudown the Raspberry Pi.

3.3.13

4 Developers Guide

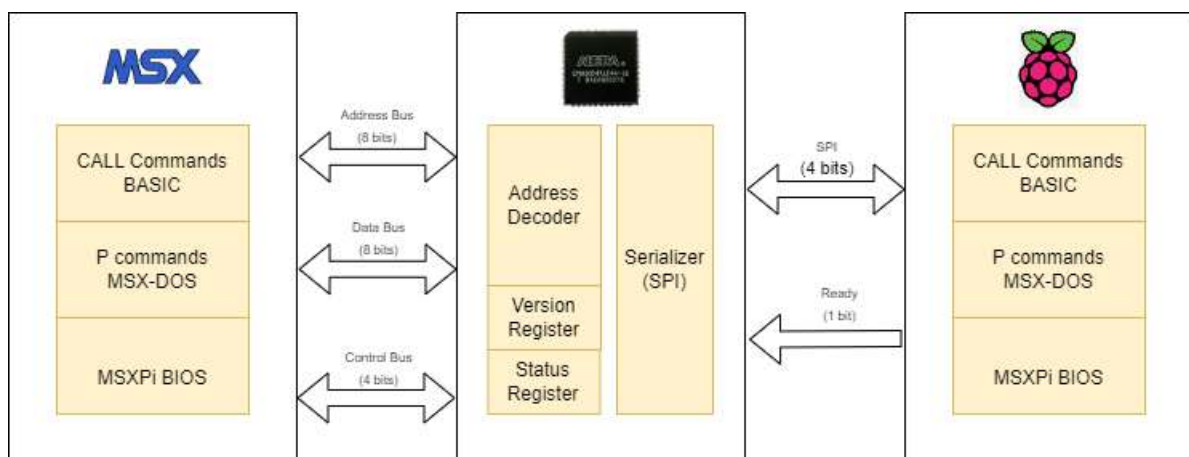
This section is directed to developers that want to create new MSXPi programs, in either BASIC or Assembly in the MSX side. Note that all code on the server side is always Python.

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, 0x57 and 0x5A. 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 commended method to develop for MSXPi is to use the BIOS and routines available. These routines implements all the low level aspects fo the interface, making it easy 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 in the program. This function must have been implemented, otherwise the server will return an error.

Data is transfered in blocks of size 8,128,, 256,or 512 bytes, depending pn the stage and function being implememnted. Its posible to use blocks of any size when calling the basic transfer routines directly (SENDDATAT and RECVDATA).

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

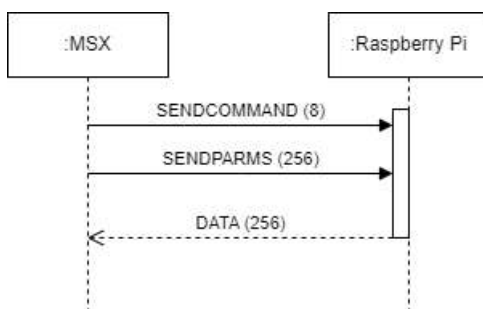
Once the function is called, it's up to the programmer to implement the required behaviour, which may include passing additionnal parameters, sending and receiving data on both directions, etc. The MSXPi BIOS routines have a number of functions the developer can use to implement their programs, which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

Most of the MSXPi commands follow this sequence:

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 comamnds (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.

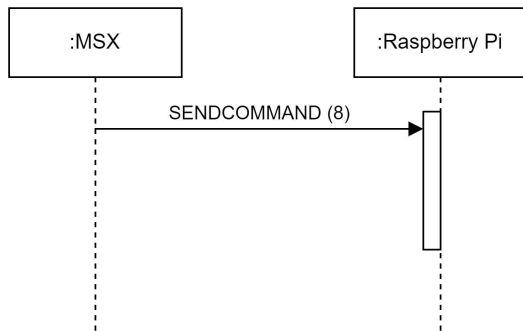


MSXPiProtocol 1

Send a command, followed by the parametes. 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 sucessful or failure situations:

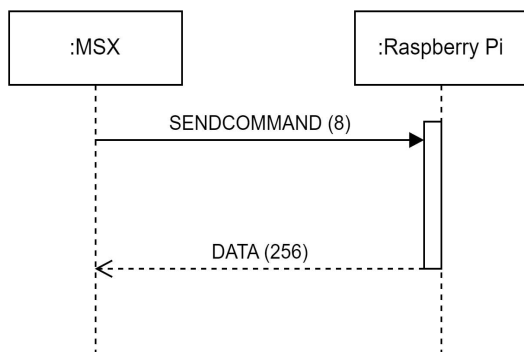
- Output of the command executed in the Raspberry Pi
- Error message when the command failed

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.



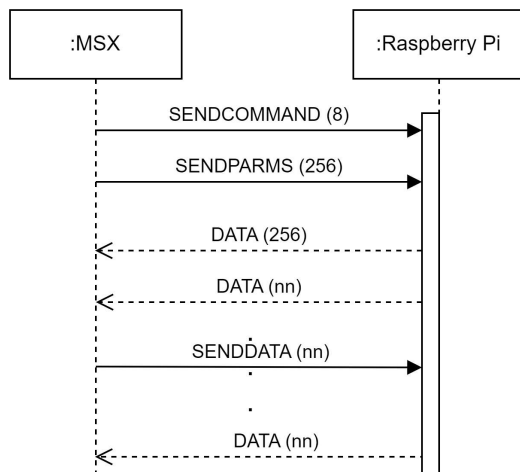
MSXPIProtocol 2

Send a command, and do not expect a response.



MSXPIProtocol 3

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



MSXPiProtocol 4

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. 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.

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 data 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 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 RPi – 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 RPi. 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 though the transfers occur in blocks of fixed size (BLKSIZE), the actual user data might be less than 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 contains the size of the useful data in that block. The client program must control how much data and blocks are 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 "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 needs 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 no valid data in the buffer.
RC_SUCCESS	Operation successful. This code is returned by the RPi. There are no blocks to be transferred. Buffer contains valid data.
RC_FAILED	Operation failed. This code is returned by RPi. There are no more blocks to be transferred. Buffer may contain a valid error message.
RC_READY	Operation successful. This code is returned by the RPi. There are more blocks to be transferred. Buffer contains 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: RPi 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 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 succeeded. The return code is RC_SUCCESS, and there will be data available in the buffer.
4. RPi received the command, processed and it succeeded. The return code is RC_READY, meaning there is another block 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 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.

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 RPi – 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 RPi 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 RPi and print on screen. For this reason, it 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$(RC) ' Get Return Code
30 IF RC = &HE0 THEN GOSUB 200 ' Print the data received from Raspberry Pi (assuming is text)
50 END
99 ' Receive some data from RPi
100 CALL MSXPIRECV("D000") ' Received data from Raspberry Pi
110 RETURN
200 S=PEEK(&HD001)+256*PEEK(&HD002) ' Get size of useful data in buffer
210 FOR M = 0 TO S-1 ' Will read all bytes in the buffer
220 PRINT CHR$(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 many others are available and can be used by the developer.

The template below shows 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.com”](https://msxpi.software/client/src/template.asm.com)

Before starting writing the program, refer to section 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 Bytes (BLKSIZE)
ld    de,buf
ld    bc,BLKSIZE
call  CLEARBUF

; Check if there are parameters in the command and send to Raspberry Pi
call  SENDPARMS
...
...
ret
command: db "pdir",0
```

4.4.1.3 SENDDATA

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
```

4.4.1.4 RECVDATA

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
```

4.4.1.5 PRINTPISTDOUT

4.4.1.6 PRINTNLIN

4.4.1.7 PRINT

4.4.1.8 CLEARBUF