AxiCat Server

v1.3.0

User Manual

December 2014

Table of Contents

Revision History

Date	Authors	Description
2014-09-05	Peter S'heeren	Initial release.
2014-09-17	Peter S'heeren	Added 1-Wire commands.
2014-09-28	Peter S'heeren	Added 1-Wire enumeration.
2014-12-12	Peter S'heeren	Added 1-Wire probing.

1 Distribution

An executable for each supported platform is distributed in separate package.

The full source of the program is distributed as part of the AxiCat software packages **axicat.tar.gz** and **axicat.zip**.

2 Installation

1. Download the package for your platform, for example **axicatserver-1.2.0-linux-armel.tar.gz**.
2. Extract the downloaded package in a local directory.
3. You can now execute the program. Run without command line parameters to display help.

3 Program

Overview

The AxiCat Server is an important tool for working with the AxiCat. The program is usually run as a server process hence the name. It offers the functionality of the AxiCat to its clients.

The server offers two modes of communication with the client:

- Network mode: a socket interface enables communication over the network.
- Standard I/O mode: another process communicates with the AxiCat by means of redirecting its input and output to the AxiCat server.

The server uses a simple client protocol that is both human-readable and easy to format and parse programmatically. Commands and responses are composed of ASCII characters.

Commands can be padded with spaces to enhance readability. This provision is very convenient when one uses the server interactively by means of a terminal program, for example PuTTY.

Empty lines and comments are permitted as well. These features come in handy when you create script files.

The server is based on the AxiCat Application Layer and offers the same high performance for all available I/O. You can easily schedule multiple transfers for GPIO, I2C, SPI and UARTs concurrently. The server will efficiently handle all transfers with the AxiCat and report appropriate responses when transfers have been completed. Cancellation of transfers is supported.

Command Line

Parameter	Description
-h	Display help and exit.
-console	Open a console in Windows.
-v	Enable verbose output.
-p n	Select network mode. The value specifies the port number the server must listen to. Value n=1..65535 decimal.
-crlf	Conclude responses with CR→LF instead of LF. The parameter only applies when network mode is selected.
-stdio	Select standard I/O mode.
-s PATH	Select the AxiCat with the given serial path as the interface. Example PATH in Linux: /dev/ttyUSB0 Example PATH in Windows: \\.\COM4
-i FILE	Specify a file with initialization commands.

If no parameters are provided, the program displays help and exits.

Either **-p** or **-stdio** must be specified to select the mode of communication.

The **-crlf** parameter alters the end-of-line character sequence of responses when the server is operating in network mode. By default, the server concludes each response with a LF character. This renders undesirable output in terminal programs that distinguish between LF and CR. PuTTY, for example, doesn't move the cursor to the beginning of the line when it receives LF. The **-crlf** parameter solves this particular problem.

If parameter **-i** is specified, the server will process the given file after it's connected to the AxiCat and before it starts accepting commands from the client. Using this parameter, you can have to server send various commands to your AxiCat. Doing so enables you to bring the AxiCat to a well-defined state before any client connection can start communicating with the AxiCat.

Network Mode

The server creates the specified port when it has successfully connected to the AxiCat. If the AxiCat is disconnected, the port will be removed along with the client connection if present. Nonetheless the server will remain active and keep trying to reconnect to the AxiCat. This behavior corresponds very well with the plug-and-play nature of the USB-based AxiCat.

This also means that the AxiCat is initialized only once, during connecting, independently from connectivity on the server port. As such a client can connect with and disconnect from the server multiple times without having to worry about the state of the AxiCat; the state will not change in-between socket connections, unless the AxiCat went through a USB replugging phase. This is especially interesting if you want to communicate with the AxiCat from a website where each page has to reconnect with the server program.

The server works with one AxiCat, hence it will accept one incoming socket connection. If you want to provide access to more than one AxiCat from the same computer, you can run multiple instances of the server program, one instance for each AxiCat.

Standard I/O Mode

This mode allows another running process to directly write commands to the server and read responses back on the same system.

Typical use-cases include manually typing in and sending commands from the console, piping files with commands to the server, and controlling the AxiCat from a scripting language.

Windows Console

The Windows version of the server is built as a Win32 application, as opposed to a console application. Nonetheless, the server is capable of opening a dedicated console for displaying information. Parameters **-console** and **-h** will produce the console.

It's not advisable to combine parameter **-console** with I/O redirecting. For example:

```
> axicatserver -s \\.\COM3 -console -stdio < commands.txt > responses.txt
```

The console will take over standard input and output and the result won't be what you expect. Instead, run the command as follows:

```
> axicatserver -s \\.\COM3 -stdio < commands.txt > responses.txt
```

No console will appear and the server will perform the job as expected, taking in the commands from the input file and emitting the responses to the output file.

Usage Examples

Network Mode and Terminal Programs

Suppose you want the server to enable access to the AxiCat via network port 4000. The following command will do the job in Linux:

```
$ ./axicatserver -s /dev/ttyUSB0 -p 4000 &
```

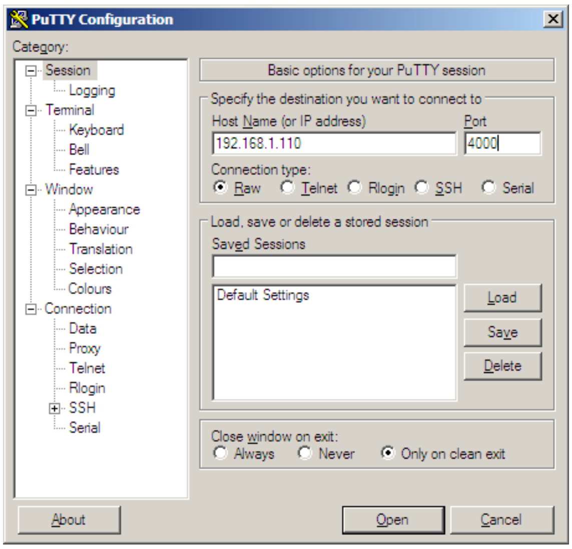
This will run the server silently in the background.

The equivalent command in Windows is:

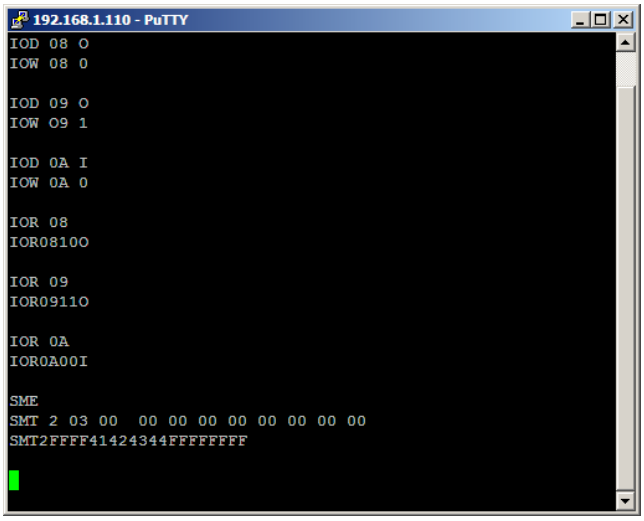
```
> axicatserver -s \\.\COM10 -p 4000
```

The server will start listening on port 4000 as soon as it's connected to the AxiCat. Any computer on your network that has access to the server is now able to communicate with the AxiCat.

You can manually send commands using a terminal program, like PuTTY. For example:



Host Name (or IP address): 192.168.1.110
Port: 4000
Connection type: ● Raw ○ Telnet ○ Rlogin ○ SSH ○ Serial



```
IOD 08 O
IOW 08 0

IOD 09 O
IOW 09 1

IOD 0A I
IOW 0A 0

IOR 08
IOR08100

IOR 09
IOR09110

IOR 0A
IOR0A00I

SME
SMT 2 03 00  00 00 00 00 00 00 00 00
SMT2FFFF41424344FFFFFFFF
```

In the above example, the server is connected to from a remote computer. All it takes is the IP address and port number of the server. Next, a number of commands are typed in to which the server responds.

Another convenient way of connecting to the server is using **netcat** in linux:

```
$ nc 192.168.1.110 4000
SME
SMT 2 03 00  00 00 00 00 00 00 00 00
SMT2FFFF41424344FFFFFFFF
```

For the curious, the SPI transfer actually reads out the first 8 bytes from a Microchip 25LC020A EEPROM chip. The $\overline{\text{CS}}$ pin of the EEPROM is connected to $\overline{\text{SS2}}$ on the AxiCat. The first four bytes had been programmed as "ABCD" earlier.

Network Mode and Programming Languages

Network mode is great for controlling the AxiCat from a programming language. Every popular language has built-in support for programming with network sockets. The format of commands and responses allows for very simple processing in your language of choice.

Run the server in network mode. This command will do the job in Linux:

```
$ ./axicatserver -s /dev/ttyUSB0 -p 4000 &
```

The equivalent command in Windows is:

```
> axicatserver -s \\.\COM10 -p 4000
```

The following example shows how to use the AxiCat from a Python program. Enter the following Python code in an editor:

```
import socket

# Change host if needed. For example: '192.168.1.110'
host = 'localhost'

# Change port number if already used.
port = 4000

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))

s.send('IOR05\n')

data = s.recv(1024)
print 'Received:'
print data

s.close()
```

Don't forget to change host if needed. Save this file as **client.py** for example.

You can run the program in Linux as follows:

```
$ python client.py
Received:
IOR0511O

$
```

The output shows that the Python program successfully connected to the server, sent the GPIO read command, and received the response containing the state of GPIO pin 5.

Standard I/O Mode and Console

A typical use for standard I/O mode is manually typing in commands in the console. Example in Linux:

```
$ ./axicatserver -s /dev/ttyUSB0 -stdio
IOR00
IOR0011I
```

To accomplish the same in Windows, a console must be opened:

```
> axicatserver -s \\.\COM10 -stdio -console
```

An empty console appears where you can type in commands. A great feature of Windows console is line editing and history; you can use the arrow keys to navigate in all four directions.

Note: if you forget the **-console** parameter, the process will start up in the background and immediately terminate.

Standard I/O Mode and Piping

Another application of standard I/O mode is piping. You can pipe an input file with commands to the server and pipe the responses to another file, like this:

```
$ ./axicatserver -s /dev/ttyUSB0 -stdio < commands.txt > responses.txt
```

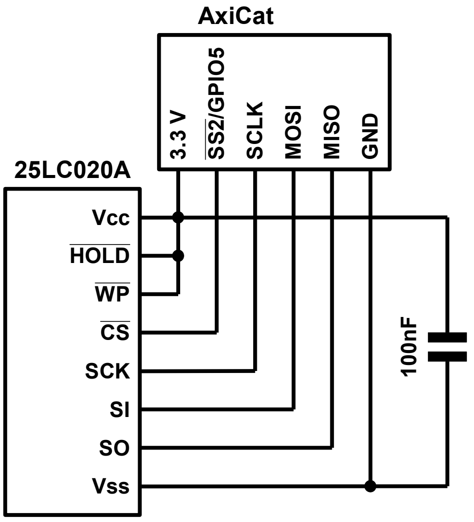
The equivalent command in Windows is:

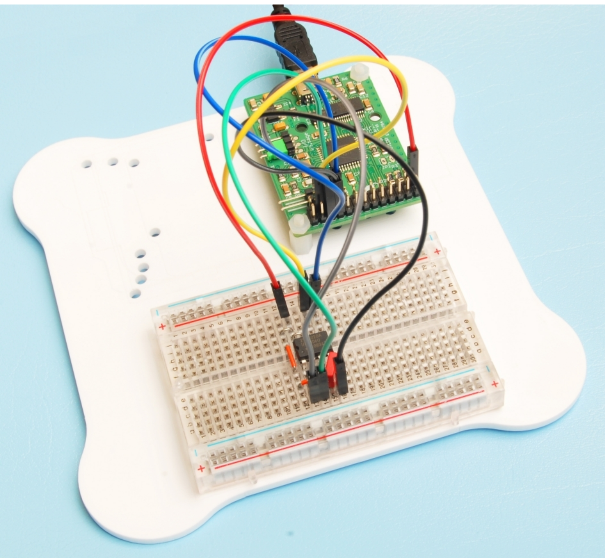
```
> axicatserver -s \\.\COM10 -stdio < commands.txt > responses.txt
```

As mentioned in section Windows Console, don't specify the **-console** parameter in this case.

Initialization File

Let's create an initialization file for the following circuit. We've taken a Microchip 25LC020A SPI-EEPROM chip and connected it to the AxiCat.





We want the server to automatically initialize the AxiCat for use with the circuit as soon as the AxiCat is connected. Initialization for this circuit requires setting up GPIO5 and enabling the SPI master.

Create the initialization file with an editor:

```
# Initialization file for SPI-EEPROM, SS2 as chip select

# Set up SS2/GPIO5 as output, logic one
IOD 05 O
IOW 05 1

# Enable the SPI master
SME
```

Save the file as **spieeprom_init.txt** for example.

Let's start the server in network mode. This command will do the job in Linux:

```
$ ./axicatserver -s /dev/ttyUSB0 -p 4000 -i spieeprom_init.txt &
```

The equivalent command in Windows is:

```
> axicatserver -s \\.\COM10 -p 4000 -i spieeprom_init.txt
```

With the server up-and-running, you can now start working with the EEPROM chip without having to worry about setting up the AxiCat. Here are some example SPI transfers containing EEPROM commands:

Command	`SMT 2 05 00`	RDSR – read status register
Response	`SMT2FF00`	Status register is 00000000b
Command	`SMT 2 06`	WREN – enable write operation
Response	`SMT2FF`	
Command	`SMT 2 05 00`	RDSR – read status register
Response	`SMT2FF02`	Status register is 00000010b
Command	`SMT 2 02 20 45 46 47 48`	WRITE – write 4 bytes at offset 20h
Response	`SMT2FFFFFFFFFFFF`	
Command	`SMT 2 03 20 FFFFFFFFFFFFFFFF`	READ – read 8 bytes at offset 20h
Response	`SMT2FFFF45464748FFFFFFFF`	

Shutting Down

When you run the program as a server process, it silently runs in the background. In order to shut down the server, you can:

- Send the quit command **QU**.
- Kill the process.

The first method only works when the server is connected to the AxiCat and the network port has been created. Remember that the server will only accept a connection to the network port when the AxiCat is present.

Killing the process always works. The procedure depends on the operating system.

Linux

Look up the process ID of the server and send the kill signal. You must have root permissions to do that. For example:

```
# ps -A | grep axicatserver
 2111 pts/0    00:00:01 axicatserver
# kill -9 2111
```

Windows

Press CTRL-ALT-DEL to open the Task Manager. Look for a process with image name **axicatserver.exe**. Right-click and choose End Process. The process will be terminated.

4 Client Protocol

Overview

The server implements the AxiCat Application Layer module in its back-end and exposes a server socket (parameter **-p**) or standard I/O (parameter **-stdio**) in its front-end.

The client is the process that communicates with the server's front-end. The client can be any kind of process, like a terminal program, a command line interpreter, a Python program.

The communications protocol between server and client is composed of commands and responses. The client sends commands to the server, the server sends responses to the client.

Some commands produce a response while others don't. A response may be without command; this a called an unsolicited response. A subset of commands execute asynchronously ("in the background") meaning their response comes back some time later, even after the server may have processed subsequent commands.

Format of Commands and Responses

The server will only execute a command if it's correctly formatted. Parameter **-v** allows you to observe processing of commands and possibly errors.

Both commands and responses are composed of a string of ASCII characters concluded by an end-of-line marker.

The server recognizes LF (10) and CR (13) each as an end-of-line marker. This means when the client concludes a command with CR→LF, the server will actually receive two lines, a line with a command followed by an empty line. This is no problem though since empty lines are simply discarded.

Each response from the server is concluded with LF or LF→CR. See parameter **-crlf** for details.

Command may be padded with space and tab characters. The server simply filters them out. These characters do not serve as delimiters.

A '#' character indicates the start of a comment. The server discard all remaining characters until the end-of-line marker is reached.

With these rules in mind, the server interprets the following example commands identically:

```
IMT4861626364
IMT 48 61 62 63 64
I M T 4 86 16 26364
IM T 48 61626364   # SLA+W, address 24h, "abcd"
```

The following formatting is used in the description of the commands and responses:

Format	Description
QU	Literal characters i.e. character 'Q' followed by character 'U'.
ABC	These characters containing specific information as explained.
[ABC]	An array of these characters containing specific information as explained.
(A)	Optional characters containing specific information as explained.
<EOL>	End-of-line marker, either LF (10) or CR (13) or LF→CR.
<none>	No response.

GPIO Write

Command	**IOW**nns**<EOL>**	
	nn	GPIO pin number, 00..10 hexadecimal.
	s	Output state, 0 or 1.

Write the output state of a GPIO pin.

Example

Command	**IOW 0E 1<EOL>**

Set output of GPIO pin 14 to one.

GPIO Read

Command	**IOR**nn**<EOL>**	
	nn	GPIO pin number, 00..10 hexadecimal.
Response	**IOR**nniod**<EOL>** **IOR**nn**SC<EOL>**	
	nn	GPIO pin number, 00..10 hexadecimal.
	i	Input (sensed) state, 0 or 1.
	o	Output state, 0 or 1.
	d	Direction, I for input, O for output.
	s	Non-successful transfer status: canceled.

Read information about a GPIO pin.

Example

Command	**IOR01<EOL>**
Response	**IOR0100O<EOL>**

Read GPIO pin 1. Input state is zero, output state is zero, direction is output.

GPIO Set Direction

Command	IODnnd<EOL>	
	nn	GPIO pin number, 00..10 hexadecimal.
	d	Direction, I for input, O for output.

Set the direction of a GPIO pin.

Example

Command	IOD 05 I<EOL>

Set GPIO pin 5 as input.

I2C Master Set Speed

Command	IMSS[n]<EOL>	
	[n]	Bus speed, one or more decimal digits.

Set the speed of the I2C bus. The given value must match one of the predefined bus speeds defined in the AxiCat protocol (see axicat.h, AXICAT_TWI_SPEED_<n>).

Example

Command	IMSS 100000<EOL>

Set the speed of the I2C bus to 100000 Hz.

I2C Master Set Raw Speed

Command	IMSRrrs<EOL>	
	rr	TWBR, 00..FF hexadecimal.
	s	TWPS, 0..3 hexadecimal.

Directly write to the speed registers of the I2C function.

Refer to section “Two-wire Serial Interface” in the Atmega164A/324A/644A/1284 datasheet for more information.

Examples

Command	IMSR3A1<EOL>

Set the bit rate register to 3Ah (58) and the prescaler register to 1, resulting in a bus speed of 25000 Hz.

Command	IMSR000<EOL>

Set the bit rate register to 00h and the prescaler register to 0, resulting in a bus speed of 750000 Hz. This is the maximum I2C speed.

I2C Master Enable

Command	**IME<EOL>**

This command enables the I2C (TWI) part of the AxiCat.

I2C Master Disable

Command	**IMD<EOL>**

This command disables the I2C (TWI) part of the AxiCat.

Note that you can schedule transfers even when the I2C master is disabled.

All transfers being processed in the AxiCat will be completed. However, one or more transfers that are still scheduled in the application layer will be initiated in the AxiCat. The AxiCat will merely buffer these transfers, but for the application layer the transfers have been started.

If your intention is to cancel all master transfers and disable the I2C function, issue the following commands:

Command	**IMC<EOL>** **IMD<EOL>**

If you want to cancel all slave transfers as well:

Command	**IMC<EOL>** **ISCW<EOL>** **ISCR<EOL>** **IMD<EOL>**

The order is important. First request cancellation, then disable the I2C function. If you do it the other way around, the application layer is capable of sending one or more scheduled transfers to the AxiCat between **IMD** and **IMC**, **ISCW**, **ISCR**.

I2C Master Transfer

This command handles both write (SLA+W) and read (SLA+R) transfers.

Write transfer:

Command	**IMT**aa([dd])(n)**<EOL>**		
	aa	Slave address and direction.	
		Bit 7..1	Slave address.
		0	Zero, indicating a write (SLA+W) transfer.
	([dd])	Zero or more data bytes, 00..FF hexadecimal.	
	(n)	Specify P to force a stop signal. Specify R for explicit repeated start.	
Response	**IMT**aannnnr**<EOL>** **IMT**aa**S**s**<EOL>**		
	aa	Slave address and direction.	
		Bit 7..1	Slave address.
		0	Zero, indicating a write (SLA+W) transfer.
	nnnn	Number of transferred bytes, 0000.. hexadecimal.	
	r	Response of the slave device, A for ACK, N for NACK. If no bytes were transferred, it's the slave's response to the SLA+W, else it's the slave's response to the last byte written by the master.	
	s	Non-successful transfer status: B for bus error, A for arbitration lost, S for skipped, C for canceled.	

Read transfer:

Command	**IMT**aannnn(n)**<EOL>**		
	aa	Slave address and direction.	
		Bit 7..1	Slave address.
		0	One, indicating a read (SLA+R) transfer.
	nnnn	Number of bytes to read from the slave device, 0000.. hexadecimal.	
	(n)	Specify P to force a stop signal. Specify R for explicit repeated start.	
Response	**IMT**aar([dd])**<EOL>** **IMT**aa**S**s**<EOL>**		
	aa	Slave address and direction.	
		Bit 7..1	Slave address.
		0	One, indicating a read (SLA+R) transfer.
	r	Response of the slave device to SLA+R, A for ACK, N for NACK.	
	([dd])	When the slave responded with ACK, one or more data bytes received from the slave, 00..FF hexadecimal.	
	s	Non-successful transfer status: B for bus error, A for arbitration lost, S for skipped, C for canceled.	

These commands initiate a I2C transfer.

If 'R' is specified, the server withholds scheduling of the transfer. As soon as a subsequent IMT command without 'R' comes in, the server will schedule all withheld transfers. Each transfers marked with 'R' will induce a repeated start for the following transfer. This way, the client can commit a list of transfers that the I2C master will execute without releasing the I2C bus.

If neither 'R' nor 'L' is specified, the AxiCat application layer will decide whether a STOP signal or REPEATED START signal will be generated at the end of the I2C transfer. If this is the last scheduled I2C transfer, the application layer concludes with a STOP signal, else it generates a REPEATED START signal and issues the next I2C transfer.

The max. number of data bytes is 65535.

Examples

Command	`IMT 50 41 42 43 44<EOL>`	SLA+W, address 28h, data bytes
Response	`IMT500004A<EOL>`	Slave accepts all data bytes

Successful transmission of data bytes from AxiCat to I2C slave.

Command	`IMT 50 41 42 43 44<EOL>`	SLA+W, address 28h, data bytes
Response	`IMT500002N<EOL>`	Slave accepts less data bytes

Partial transmission of data bytes from AxiCat to I2C slave.

Command	`IMT 50 41 42 43 44<EOL>`	SLA+W, address 28h, data bytes
Response	`IMT500000N<EOL>`	NACK response to SLA+W

Failed transmission of data bytes to I2C slave. The slave may be absent, or it may respond with NACK deliberately.

Command	`IMT 51 0004<EOL>`	SLA+R, address 28h, read 4 bytes
Response	`IMT51A61626364<EOL>`	Slave returns all data bytes

Successful transmission of data bytes I2C slave to AxiCat.

Command	`IMT 51 0004<EOL>`	SLA+R, address 28h, read 4 bytes
Response	`IMT51A6162<EOL>`	Slave returns some data bytes

Partial transmission of data bytes I2C slave to AxiCat.

Command	`IMT 51 0004<EOL>`	SLA+R, address 28h, read 4 bytes
Response	`IMT51N<EOL>`	NACK response to SLA+R

Failed transmission of data bytes to I2C slave. The slave may be absent, or it may return a NACK response deliberately.

Command	`IMT 6A 05 R   <EOL>`
	`IMT 6B 0004 R<EOL>`
	`IMT 6A 1A R   <EOL>`
	`IMT 6B 0010 P<EOL>`

Initiate various I2C transfers in one block. When the server receives the last command, it schedules all four transfers at once. As a result, the AxiCat generates a REPEATED START signal between I2C transfers. The fourth transfer is explicitly concluded with a STOP signal.

Note that in case the I2C master looses arbitration, all remaining I2C transfers in the block will be skipped.

Command	**IMT40**<EOL> **IMT42**<EOL>
Response	**IMT400000A**<EOL> **IMT420000N**<EOL>

Probe slave addresses 20h and 21h. The slave with address 20h responded with ACK, slave address 21h was not acknowledged.

I2C Master Cancel

Command	**IMC**<EOL>

Request cancellation of all master transfers.

I2C Slave Enable

Command	**ISE**aa<EOL>		
	aa	Slave address and general call address.	
		Bit 7..1	Slave address.
		0	General call address is enabled (1) or disabled (0).

Enable the I2C slave function.

When the I2C slave is enabled, it will respond to SLA+W and SLA+R packets carrying the specified slave address. When the slave is addressed, either a slave Tx transfer or a slave Rx transfer must have been scheduled to let the master continue the bus transaction. If the required transfer isn't scheduled, the I2C slave will lock up the I2C bus until the Slave Tx or Rx transfer is scheduled.

I2C Slave Disable

Command	**ISD**<EOL>

Disable the I2C slave function.

I2C Slave Write

Command	ISW[dd](L)<EOL>	
	[dd]	One or more data bytes, 00..FF hexadecimal.
	(L)	Specify L to mark as the last chunk of the data payload.
Response	ISWnnnnr<EOL> ISWSs<EOL>	
	nnnn	Number of bytes written to the master, 0000.. hexadecimal.
	r	Response of the master, A for ACK, N for NACK.
	s	Non-successful transfer status: B for bus error, S for skipped, C for canceled.

Schedule a slave Tx transfer. When an I2C master addresses the slave with SLA+R, it can read the given data bytes from the slave.

You can split up the data in chunks that form one logical data payload. This features is useful when the slave can't produce all data bytes in time. When 'L' is omitted and the I2C master has read data from the slave, the slave will lock up the I2C bus until another slave Tx transfer is scheduled. Thus it's important to schedule a slave Tx transfer with 'L' at some point. In many cases, you even don't have to split up the data payload so you'd always specify 'L'.

I2C Slave Read

Command	ISRnnnn(L)<EOL>	
	nnnn	Number of bytes to read from the master, 0001.. hexadecimal.
	(L)	Specify L to mark as the last chunk of the data payload.
Response	ISR[dd]<EOL> ISRSs<EOL>	
	[dd]	One or more data bytes read from the master, 00..FF hexadecimal.
	s	Non-successful transfer status: B for bus error, S for skipped, C for canceled.

Schedule a slave Rx transfer. When an I2C master addresses the slave with SLA+W, it can write data bytes to the slave.

I2C Slave Cancel Write

Command	ISCW<EOL>

Request cancellation of all slave Tx transfers.

I2C Slave Cancel Read

Command	ISCR<EOL>

Request cancellation of all slave Rx transfers.

SPI Master Set Speed

Command	SMSS[n]<EOL>	
	[n]	Bus speed, one or more decimal digits.

Set the speed of the SPI bus. The given value must match one of the predefined bus speeds defined in the AxiCat protocol (see axicat.h, AXICAT_SPI_SPEED_<n>).

Example

Command	SMSS 750000<EOL>

Set the speed of the SPI bus to 750000 Hz.

SPI Master Set Raw Speed

Command	SMSRcx<EOL>	
	c	CR, 0..3 hexadecimal.
	x	X2, 0..1 hexadecimal.

Set the speed registers of the SPI bus.

SPI Master Set Configuration

Command	SMSCpqo<EOL>	
	p	Clock polarity. Clock is low (0) or high (1) when idle.
	q	Clock phase. Sample on leading (0) or trailing (1) edge.
	o	Bit order is MSb→LSb (0) or LSb→MSb (1).

Configure the SPI bus.

SPI Master Enable

Command	SME<EOL>

Enable the SPI master.

SPI Master Disable

Command	SMD<EOL>

Disable the SPI master.

This command disables the SPI part of the AxiCat.

Note that you can schedule transfers even when the SPI master is disabled.

All SPI master transfers being processed in the AxiCat will be completed. However, one or more transfers that are still scheduled in the application layer will be initiated in the AxiCat. The AxiCat will merely buffer these transfers, but for the application layer the transfers have been started.

If your intention is to cancel all SPI master transfers and disable the SPI function, issue the following commands:

Command	SMC<EOL> SMD<EOL>

The order is important. First request cancellation, then disable the SPI function. If you do it the other way around, the application layer is capable of sending one or more scheduled transfers to the AxiCat between **SMD** and **SMC**.

SPI Master Transfer

Command	SMTn[dd]<EOL>	
	n	Slave select line, 0..3 hexadecimal.
	[dd]	One or more data bytes to transmit, 00..FF hexadecimal.
Response	SMTn[dd]<EOL> SMTnSs<EOL>	
	n	Slave select line, 0..3 hexadecimal.
	[dd]	One or more received data bytes, 00..FF hexadecimal.
	s	Non-successful transfer status: S for skipped, C for canceled.

This command schedules an SPI transfer.

The command and the response contain the same number of data bytes, unless the transfer was aborted. The latter may happen if the SPI function is disabled while the transfer is being executed.

The max. number of data bytes is 65535.

SPI Master Cancel

Command	SMC<EOL>

Request cancellation of all SPI transfers.

1-Wire Master Enable

Command	OME<EOL>

Enable the 1-Wire master.

1-Wire Master Disable

Command	OMD<EOL>

Disable the 1-Wire master.

This command disables the 1-Wire part of the AxiCat.

Note that you can schedule transfers even when the 1-Wire master is disabled.

All 1-Wire master transfers being processed in the AxiCat will be completed. However, one or more transfers that are still scheduled in the application layer will be initiated in the AxiCat. The AxiCat will merely buffer these transfers, but for the application layer the transfers have been started.

If your intention is to cancel all 1-Wire master transfers and disable the 1-Wire function, issue the following commands:

Command	OMC<EOL> OMD<EOL>

The order is important. First request cancellation, then disable the 1-Wire function. If you do it the other way around, the application layer is capable of sending one or more scheduled transfers to the AxiCat between **OMD** and **OMC**.

1-Wire Master Reset

Command	OMR<EOL>	
Response	OMRp<EOL> OMRSs<EOL>	
	p	One or more 1-Wire slaves are present (1) or no slave is present on the 1-Wire bus (0).
	s	Non-successful transfer status: S for skipped, C for canceled.

This command schedules a 1-Wire reset.

1-Wire Master Touch Bytes

Command	OMT[dd]s<EOL>	
	[dd]	One or more data bytes to transmit, 00..FF hexadecimal.
	s	Activate strong pull-up (1) or don't (0).
Response	OMT[dd]<EOL> OMTSs<EOL>	
	[dd]	One or more received data bytes, 00..FF hexadecimal.
	s	Non-successful transfer status: S for skipped, C for canceled.

This command schedules a 1-Wire touch bits transfer. The data bits are specified as an array of bytes. As such, the command always transfers a multiple of 8 bits.

The command and the response contain the same number of data bytes, unless the transfer was aborted. The latter may happen if the 1-Wire function is disabled while the transfer is being executed.

The max. number of data bytes is 8191.

For example, read the scratchpad of a single DS18B20 connected to the 1-Wire bus:

Command	OMR<EOL> OMT CC BE FFFFFFFFFFFFFFFFFF<EOL>	Reset Skip ROM → Read Scratchpad
Response	OMR1<EOL> OMTCCBE54014B467FFF0C10FD<EOL>	Reset, presence detected Skip ROM → Read Scratchpad

1-Wire Master Touch Bits

Command	OMB[d]s<EOL>	
	[d]	One or more data bits to transmit, 0 or 1 digit.
	s	Activate strong pull-up (1) or don't (0).
Response	OMB[d]<EOL> OMBSs<EOL>	
	[d]	One or more received data bits, 0 or 1 digit.
	s	Non-successful transfer status: S for skipped, C for canceled.

This command schedules a 1-Wire touch bits transfer.

The command and the response contain the same number of data bits, unless the transfer was aborted. The latter may happen if the 1-Wire function is disabled while the transfer is being executed.

The max. number of data bits is 65535.

For example, read the power mode of a single DS18B20 connected to the 1-Wire bus:

Command	OMR<EOL> OMT CC B4<EOL> OMB1<EOL>	Reset Skip ROM → Read Power Supply Read bit
Response	OMR<EOL> OMTCCB4<EOL> OMB0<EOL>	Reset Skip ROM → Read Power Supply DS18B20 reports parasite power

1-Wire Master Enumerate

Command	OMNF(C)(Fff)(snn-[n](-nn))<EOL> ONNN<EOL>	
	(C)	Optionally specify C to search for slaves in the alarmed state instead of all slaves. This is also known as alarm condition search.
	(Fff)	Optionally specify F followed by a family code (00..FF hexadecimal) to narrow down the enumeration.
	(s…)	Optionally enable DS2409 smart ON: M for MAIN, A for AUX, followed by a ROM code that specifies the DS2409.
Response	OMN(nn-nnnnnnnnnnnn-nn)<EOL> OMNs<EOL>	
	(…)	Enumerated ROM code.
	s	Non-successful transfer status: S for skipped, C for canceled.

The OMN command schedules one iteration of a 1-Wire enumeration. The command either initiates an enumeration procedure (OMNF – First) or continues the procedure (OMNN – Next). Both variations of the command generate a response of the same format.

OMNF establishes a new enumeration context based on the specified search criteria, and executes the first iteration of the enumeration procedure. In other words, OMNF searches for the first 1-Wire slave.

OMNN continues from where the previous OMNF or OMNN command left and searches for the next 1-Wire slave, if any.

OMNF without parameters will start a search for all 1-Wire slaves that are visible on the 1-Wire bus. You can narrow down the enumeration by specifying any combination of the following search criteria:

- Alarm condition search: Only 1-Wire slaves that have an alarm condition are enumerated.
- Family search: Only 1-Wire slaves bearing the specified family code are enumerated.
- DS2409 smart ON: If you choose this option, only 1-Wire slaves that reside behind the main or auxiliary port are enumerated. You have to choose between main and auxiliary port and you've to provide the ROM code of the target DS2409.

The DS2409 ROM code is made up of hexadecimal digits and one or two hyphens:

- The family code comes first, 00..FF hexadecimal.
- An hyphen follows.
- The serial number is specified as one or more hexadecimal digits.
- Optional:
  - An hyphen.
  - Followed by a CRC value, 00..FF hexadecimal.

The server calculates the CRC value of the DS2409 ROM code. If a CRC value is specified, it overwrites the calculated value.

Example commands:

Command	**OMNF<EOL>**	Enumerate all 1-Wire slaves
Command	**OMNF C<EOL>**	Search slaves with alarm condition
Command	**OMNF F28<EOL>**	Only enumerate DS18B20 slaves
Command	**OMNF M 1F-56BA7<EOL>**	Search behind DS2409 main port
Command	**OMNF A 1F-56BA7-80<EOL>**	DS2409 auxiliary port, CRC specified
Command	**OMNF C F28 M 1F-56BA7<EOL>**	All search criteria combined
Command	**OMNN<EOL>**	Search next slave

Example responses:

Response	**OMN01-000016A944DC-D3<EOL>**	Slave found
Response	**OMN<EOL>**	No slave found

Both command and response use a readible formatting of the ROM code, for example “28-0000040CBBB2-C4”. The actual ROM code bytes that are transferred over the 1-Wire bus are ordered as follows:

- Family code: 1 byte.
- Serial number: 6 bytes, LSB→MSB.
- CRC value, 1 byte.

ROM code “28-0000040CBBB2-C4” is transferred over the 1-Wire bus as the following byte sequence: 28h→B2h→BBh→0Ch→04h→00h→00h→C4h.

When you acquire a ROM code from an OMNN response and you want to use the ROM code bytes for addressing the 1-Wire slave in an OMT command, you'll have to flip the middle six bytes first. In the following example, we enumerate a DS18B20 and read the scratchpad data:

Command	OMNF<EOL> OMNN<EOL> OMNN<EOL>	Enumeration commands
Response	OMN28-0000040CBBB2-C4<EOL> OMN01-000016A944DC-D3<EOL> OMN<EOL>	Found a DS18B20 and a 1-Wire identification chip.
Command	OMR<EOL> OMT 55 28 B2 BB 0C 04 00 00 C4<EOL> OMT BE FF FF FF FF FF FF FF FF FF<EOL>	Reset Match ROM Read Scratchpad
Response	OMR1<EOL> OMT5528B2BB0C040000C4<EOL> OMTBE50014B467FFF101049<EOL>	Reset, presence detected Match ROM Read Scratchpad

Since the enumeration command is asynchronous, you can increase the speed of enumeration significantly by scheduling enumeration commands in advance. For example, let's schedule eight commands in one go:

Command	OMNF<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL>
Response	OMN28-0000040CBBB2-C4<EOL> OMN01-000016A944DC-D3<EOL> OMN1F-000000056B3E-A7<EOL> OMN1F-000000056B31-83<EOL> OMN1F-000000056BA7-80<EOL> OMN<EOL> OMN<EOL> OMN<EOL>

In the above example, five 1-Wire slaves are present on the bus and their ROM codes have been reported. The last three responses carry no ROM code meaning all slaves have been enumerated.

A more advanced enumeration algorithm would involve scheduling a set of commands initially (one OMNF and several OMNN commands), and then scheduling new OMNN commands as OMNnn-nnnnnnnnnnnn-nn responses come in, with the intent to keep the enumeration mechanism going.

Command	OMNF<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL>
Response	OMN28-0000040CBBB2-C4<EOL> OMN01-000016A944DC-D3<EOL> OMN1F-000000056B3E-A7<EOL> OMN1F-000000056B31-83<EOL>
Command	OMNN<EOL> OMNN<EOL> OMNN<EOL> OMNN<EOL>
Response	OMN28-0000040CBBB2-C4<EOL> OMN<EOL> OMN<EOL> OMN<EOL>

In the above example, we schedule eight enumeration commands to kick off the enumeration procedure. For every four enumerated 1-Wire slaves we schedule four new OMNN commands.

If you need very fast enumeration and the connection with the server is rather slow, then you can easily increment the number of commands to compensate for network delays. For example, you could start off with 32 commands and schedule 16 more for every 16 enumerated 1-Wire slaves.

1-Wire Master Probe

Command	OMPnn-[n](-nn)<EOL>	
	nn…	ROM code of the 1-Wire slave to probe.
Response	OMPf<EOL> OMPs<EOL>	
	f	The 1-Wire slave has been found (1) or not (0).
	s	Non-successful transfer status: S for skipped, C for canceled.

The OMP command schedules a 1-Wire probing command.

Examples

Command	OMP 28-40CBBB2<EOL>
Response	OMP1<EOL>
Command	OMP 3B-000000183368-2E<EOL>
Response	OMP0<EOL>

Since the probing command is asynchronous, you can increase the speed of probing multiple slaves significantly by scheduling commands in advance. For example:

Command	OMP 28-40CBBB2<EOL> OMP 3B-183368<EOL>
Response	OMP1<EOL> OMP0<EOL>

1-Wire Master Cancel

Command	OMC<EOL>

Request cancellation of all 1-Wire transfers.

UART Set Baudrate

Command	UuSB[n]<EOL>	
	u	Selected UART, 0 or 1.
	[n]	Baud rate, one or more decimal digits.

Set the baud rate of the selected UART.

The given value must match one of the predefined baud rates defined in the AxiCat protocol (see axicat.h, AXICAT_UART_BAUDRATE_<n>).

Example

Command	U0SB 9600<EOL>

Set the baud rate of UART0 to 9600.

UART Set Raw Baud Rate

Command	UuSRrrrx<EOL>	
	u	Selected UART, 0 or 1.
	rrr	UBRR, 000..FFF hexadecimal.
	x	X2, 0..1 hexadecimal.

Set the baud rate registers of the selected UART.

UART Set Data Bits

Command	UuSD[n]<EOL>	
	u	Selected UART, 0 or 1.
	[n]	Data bits, one or more decimal digits.

Set the number of bits per data word for the selected UART.

The given value must match one of the values defined in the AxiCat protocol (see axicat.h, AXICAT_UART_DATA_BITS_<n>).

Example

Command	U1SD9<EOL>

Set the data word size of UART1 to 9 bits.

UART Set Stop Bits

Command	UuSS[n]<EOL>	
	u	Selected UART, 0 or 1.
	[n]	Stop bits, one or more decimal digits.

Set the number of bits per data word for the selected UART.

The given value must match one of the values defined in the AxiCat protocol (see axicat.h, AXICAT_UART_STOP_BITS_<n>).

Example

Command	U1SS2<EOL>

Set 2 stop bits per data word for UART1.

UART Set Rx Timeout

Command	UuST[n]<EOL>	
	u	Selected UART, 0 or 1.
	[n]	Rx timeout, one or more decimal digits, 0..65535 milliseconds.

Set the timeout value in milliseconds for reporting received data. A value of zero means that the AxiCat will immediately report any received data word. A non-zero value programs a timeout for reporting all buffered data words since the last data word was received.

Note that the AxiCat has a limited buffer for receiving data words and will report the data anyway when the buffer is full.

Example

Command	U0ST100<EOL>

Set the Tx timeout to 100 milliseconds for UART0.

UART Set Unsolicited Rx Response

Command	UuSUnnnn<EOL>	
	u	Selected UART, 0 or 1.
	nnnn	Maximum number of data bytes to report (0..65535).

Set the maximum number of data bytes an unsolicited Rx response may report. If this value is zero, unsolicited Rx responses are disabled, else they're enabled.

The number of data bits determines the words size. If 9 data bits are selected, 2 data bytes for each word, ordered LSB→MSB, are reported by the AxiCat. So if you've set the UART to use 9 data bits, you better specify an even number of data bytes.

Example

Command	`U0SU0120<EOL>`

Enable unsolicited Rx responses for UART0. The maximum number of reported data bytes is 120h (288).

UART Enable

Command	`UuE<EOL>`	
	u	Selected UART, 0 or 1.

Enable the selected UART.

UART Disable

Command	`UuD<EOL>`	
	u	Selected UART, 0 or 1.

Disable the selected UART.

UART Write

Command	`UuW[dd]<EOL>`	
	u	Selected UART, 0 or 1.
	[dd]	One or more data bytes to transmit, 00..FF hexadecimal.

Transmit data words over the given UART's TXD line.

The number of data bits determines the words size. If 9 data bits are selected, you've to provide 2 data bytes for each word, in LSB→MSB order.

Example

Command	`U0W414242410D0A<EOL>`

Send 6 characters over the TXD line of UART0. The characters are “ABBA”→CR→LF.

UART Read

Command	`UuRnnnn<EOL>`	
	u	Selected UART, 0 or 1.
	nnnn	Maximum number of data bytes to read (1..65535).
Response	`UuR[dd]<EOL>`	
	u	UART, 0 or 1.
	[dd]	Zero or more received data bytes, 00..FF hexadecimal.

Read data from the given UART.

Response	**UuR[dd]<EOL>**	
	u	UART, 0 or 1.
	[dd]	One or more received data bytes, 00..FF hexadecimal.

Unsolicited UART Rx response.

The server reports data words received over the given UART's RXD line.

The number of data bits determines the words size. If 9 data bits are selected, 2 data bytes for each word, ordered LSB→MSB, are reported by the AxiCat. So if you've set the UART to use 9 data bits, you better specify an even number of data bytes.

Example

Response	**U1R61626364<EOL>**

UART1 received 4 characters “abcd”.

Quit

Command	**QU<EOL>**

Quit the server.

Client Protocol vs. Application Layer

The server translates between client protocol and AxiCat application layer. The server implements a one-to-one translation between front-end and back-end where possible, but some things just can't be passed through as-is.

By design the transfer commands in the client protocol can't offer all features of their counterparts in the application layer and that's where the differences are:

- Completion of transfers: the application layer supports completion of transfers in random order, the client protocol completes transfers in the order they were started.
- Cancellation of transfers: the application layer can cancel individual transfers, while the client protocol cancels all transfers of the same type (table) at once.

Transfers

Types

Both client protocol and application layer know six types of transfers:

- GPIO transfer.
- I2C master transfer.
- I2C slave Tx transfer.
- I2C slave Rx transfer.
- SPI master transfer.
- 1-Wire master transfer.

The server translates transfers between client protocol and application layer in a one-to-one fashion.

The AxiCat itself also works with these types of transfers, albeit in the form of more elementary commands that are cached in buffers of small sizes limited by the memory of the microcontroller.

Tables

The server maintains a table for each transfer type. A transfer table is a cache for transfer objects created in the application layer and for associated data buffers. A slot caches a single transfer object and its data buffer.

A transfer table keeps track of the order in which transfers are scheduled; it ensures all transfers will be reported back in the order they were sent by the client.

Preparation

When the server receives a transfer command from the client, it first prepares a slot in the transfer table. The transfer object and data buffer in the slot are set up to the point that the transfer can be scheduled in the application layer.

For example, when the client sends command **IMT2041424344**, it is first determined that the command represents an I2C master transfer. The server then picks an available slot, sets up the transfer object for a write (SLA+W) operation to the I2C slave at address 10h, and fills the data buffer with the bytes (41h, 42h, 43h, 44h). The transfer is now ready for scheduling in the application layer.

Scheduling may be delayed until one or more subsequent slots have been prepared in the table. This is the case for certain I2C master transfer commands, **IMT210004R** for example. The trailing 'R' instructs the server to withhold the transfer from scheduling until another I2C master transfer without 'R' comes in.

Scheduling

When the server schedules a transfer object in the application layer, it calls one of the scheduling API functions, like AXICAT_AL_SPI_Xfr_Schedule(). Once the call has been made, the transfer becomes part of another world where the application layer communicates with the AxiCat using the AxiCat protocol.

Completion

For each transfer table, the server checks for completion of scheduled transfers in the order they appear in the table. If the first scheduled transfer has completed, the server formats a response and sends it to the client. The slot of a completed transfer is made available for reuse, thus it caches a transfer object and a data buffer to be used when a future client transfer command comes in.

The actual completion of a transfer is handled in the application layer. A transfer may be completed for several reasons, such as successful execution or cancellation.

Cancellation

The client may request cancellation of all transfers of a transfer type. For example, command **OMC** requests the cancellation of all 1-Wire transfers. The server translates such client command into one or more calls to the application layer, one call for each scheduled transfer in the table of the target transfer type.

5 Serial Paths

You need to specify a serial path in order to communicate with a serial port. The next sections explain in more detail how you specify serial paths in each supported operating system.

Linux

Serial ports are accessible in the device directory structure. A serial path starts with **/dev**. A serial path is case-sensitive.

The following table summarizes serial paths that are commonly found on Linux systems:

Serial Path	Serial Port
/dev/ttyS0	The computer's 1st on-board serial port
/dev/ttyS1	The computer's 2nd on-board serial port
/dev/ttyS2	The computer's 3rd on-board serial port
/dev/ttyS3	The computer's 4th on-board serial port
/dev/ttyUSB0	1st USB serial adapter
/dev/ttyUSB1	2nd USB serial adapter
/dev/serial/by-id/	This directory contains symbolic links to serial devices. Each symbolic link name identifies a specific device.
/dev/serial/by-path/	This directory contains symbolic links to serial devices. Each symbolic link name represents a hardware path to the device, like a specific USB port on your computer.

Here are some useful commands you can run to get information about present serial devices and their corresponding serial path. The following commands were run on a Linux system with one on-board UART and one connected USB serial adapter.

Filter information from the kernel message buffer:

```
$ dmesg | grep 'tty'
[    0.000000] console [tty0] enabled
[    0.516785] serial8250: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
[    0.517463] 00:0b: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
[    0.559097] tty tty55: hash matches
[   66.076326] usb 2-1: FTDI USB Serial Device converter now attached to ttyUSB0
```

Use the **setserial** command to produce a list of serial paths:

```
$ setserial -g /dev/ttyS* /dev/ttyUSB*
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
/dev/ttyS1, UART: unknown, Port: 0x02f8, IRQ: 3
/dev/ttyS2, UART: unknown, Port: 0x03e8, IRQ: 4
/dev/ttyS3, UART: unknown, Port: 0x02e8, IRQ: 3
/dev/ttyUSB0, UART: unknown, Port: 0x0000, IRQ: 0, Flags: low_latency
```

List serial devices in the device directory:

```
$ ls -l /dev/ttyS* /dev/ttyUSB*
crw-rw---- 1 root dialout   4, 64 2012-02-22 14:19 /dev/ttyS0
crw-rw---- 1 root dialout   4, 65 2012-02-22 14:19 /dev/ttyS1
crw-rw---- 1 root dialout   4, 66 2012-02-22 14:19 /dev/ttyS2
crw-rw---- 1 root dialout   4, 67 2012-02-22 14:19 /dev/ttyS3
crw-rw---- 1 root dialout 188,  0 2012-02-22 14:20 /dev/ttyUSB0
```

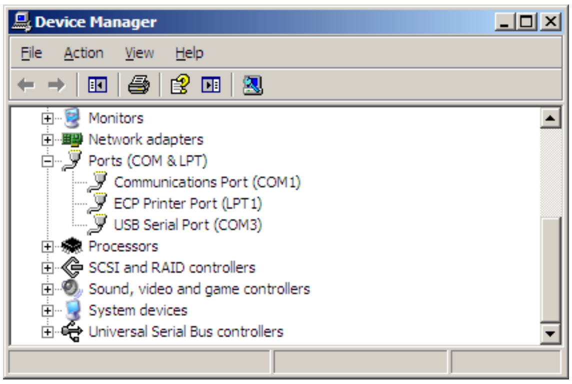
Windows

Serial ports are accessible through the Win32 device namespace, which is part of the NT namespace. As such a serial path starts with **\\.** followed by the device name of the serial port. A serial path is case-insensitive.

A serial port is typically named **COM<x>** where **<x>** is a number between 1 and 256. Other naming schemes may apply and aliases may exist, depending on the serial driver that controls the serial port.

You can obtain a list of available serial ports in the device manager. Open the device manager and view devices by type. The section named *Ports (COM & LPT)* contains all available serial and parallel ports.

The device name of a serial port is shown between parentheses. In the picture to the right, you can see two serial ports. COM1 is the PC's on-board serial port, COM3 represents a USB serial adapter.



The serial paths to the serial ports in the picture are:

Serial Path	Serial Port
\\.\COM1	*Communications Port (COM1)*
\\.\COM3	*USB Serial Port (COM3)*

Serial paths like **COM1** (that's without the **\\.** prefix) will work because COM1 to COM9 are reserved names in the NT namespace. COM10 to COM256 aren't reserved names and you'll have to specify the **\\.** prefix with these device names. By comparison, serial path **\\.\COM100** will work but serial path **COM100** won't work.

6 Software Revision History

Version	Description
1.0.0	▪ Initial release of AxiCat Server. ▪ Based on AxiCat AL v1.0.0.
1.1.0	▪ Added 1-Wire commands. ▪ Based on AxiCat AL v1.1.0.
1.2.0	▪ Added 1-Wire enumeration command. ▪ Based on AxiCat AL v1.2.0.
1.3.0	▪ Added 1-Wire probing command. ▪ Based on AxiCat AL v1.3.0.

7 Software License

The license is stated in the source code.

8 Legal Information

Disclaimer

Axiris products are not designed, authorized or warranted to be suitable for use in space, nautical, space, military, medical, life-critical or safety-critical devices or equipment.

Axiris products are not designed, authorized or warranted to be suitable for use in applications where failure or malfunction of an Axiris product can result in personal injury, death, property damage or environmental damage.

Axiris accepts no liability for inclusion or use of Axiris products in such applications and such inclusion or use is at the customer's own risk. Should the customer use Axiris products for such application, the customer shall indemnify and hold Axiris harmless against all claims and damages.

Trademarks

All product names, brands, and trademarks mentioned in this document are the property of their respective owners.

9 Contact Information

Official website: http://www.axiris.eu/