Merlin Systems Corp. Ltd

Miabot PRO BT v2

User Manual

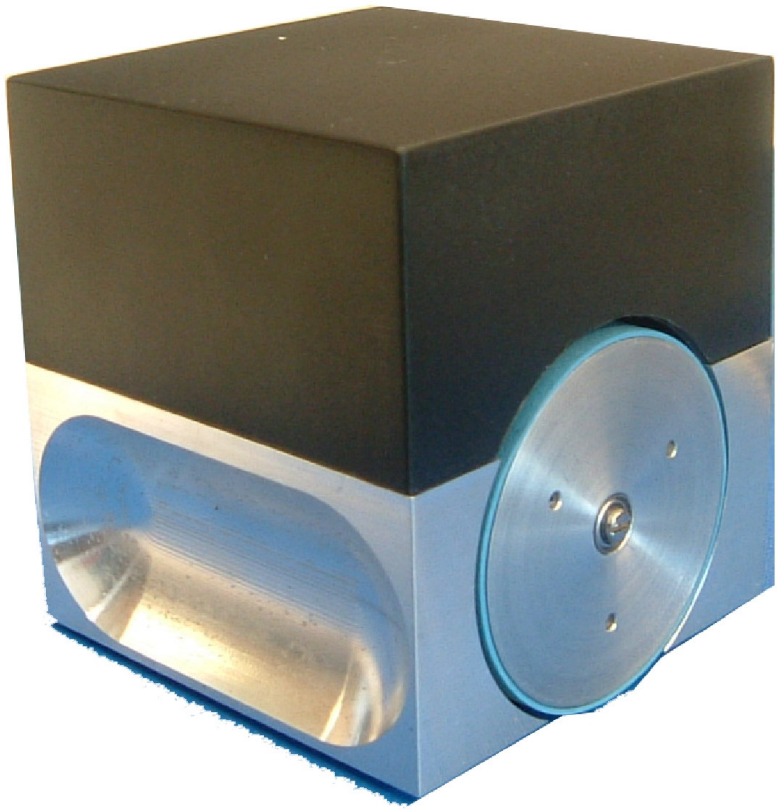
Rev. 1.3

Revision History

V1.0	24/08/04	pp	first version
V1.1	25/10/04	pp	update for v2 robot
V1.2	03/12/04	pp	renamed “UM”, added appendices A,B,C
V1.3	01/03/05	pp	added TOC, uprated apps and dev sections

Introduction

The MIABOT Pro is a fully autonomous miniature mobile robot. The latest BT version features bi-directional Bluetooth communications, which provides a robust frequency hopping wireless communications protocol at 2.4GHz.



MIABOT Pros are ideal robots for use as part of technology class tutorials, for research and development, and for high-speed and agility requirements such as robot competitive events. Universities already use MIABOTs worldwide for a wide variety of applications including FIRA robot soccer competitions, intelligent behaviours, robot swarming experiments and mobile robot navigation experiments. Miabots can also be employed as super mice, for line/maze following experiments etc.

Quick Start

Each robot is supplied with a built in demonstration program: Turn off, remove the lid and set all the DIPswitches at the back of the robot to the ON position: Then make sure robot has plenty of space and turn it on. The robot performs a programmed 'test sequence' of movements.
Remember to reset the switches so the robot does not start the test sequence next time it is turned on!

You can also drive the robot directly using the command protocol (see later section) or one of the supplied example programs such as Bigtrack BT.

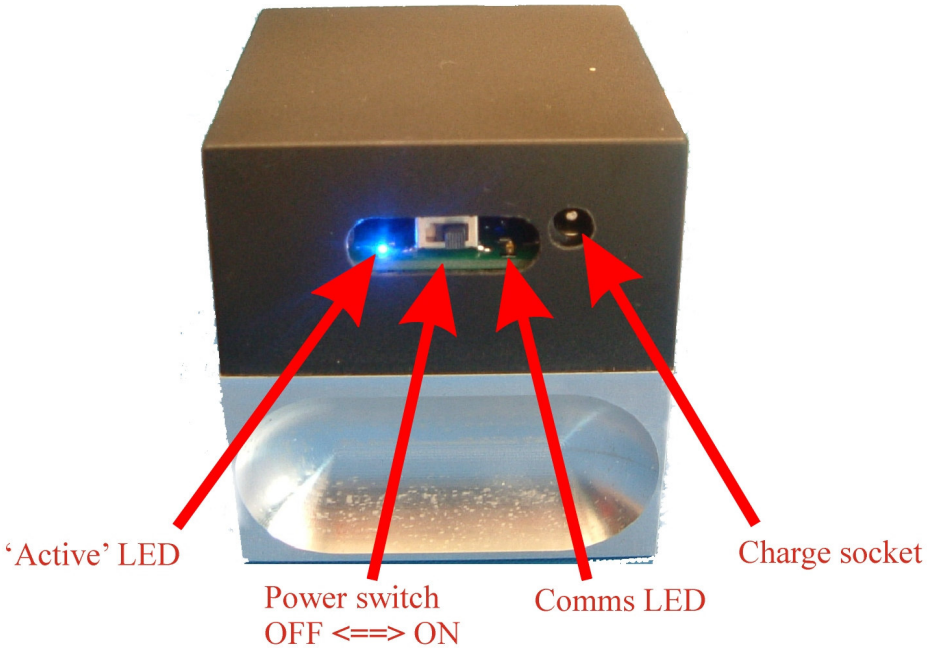
Specification

Processor	Atmel ATMega64
Speed	14.5 MIPS (RISC)
Program Memory	64Kb Flash
Data Memory	4Kb SRAM
Non-volatile storage	2Kb EEPROM
Features	In System Programmable
	Self Programming Feature (useful for adaptive learning)
	Hardware Multiply
	JTAG ICE port
Expansion Port	8 User I/O or 10 Bit A/D
Drive Train	Two wheel, Optical encoder resolution of 0.04mm
Speed	3.5m/s
Communications	Bluetooth, 11.5Kb/sec
Batteries	6 x 1.2V NiMH, rechargeable
Development	Standard toolset includes GCC 'C' compiler & linker
Simulator	FIRA Simulation Engine

Tour of Hardware

Rear View

(All connections and controls are at the rear)



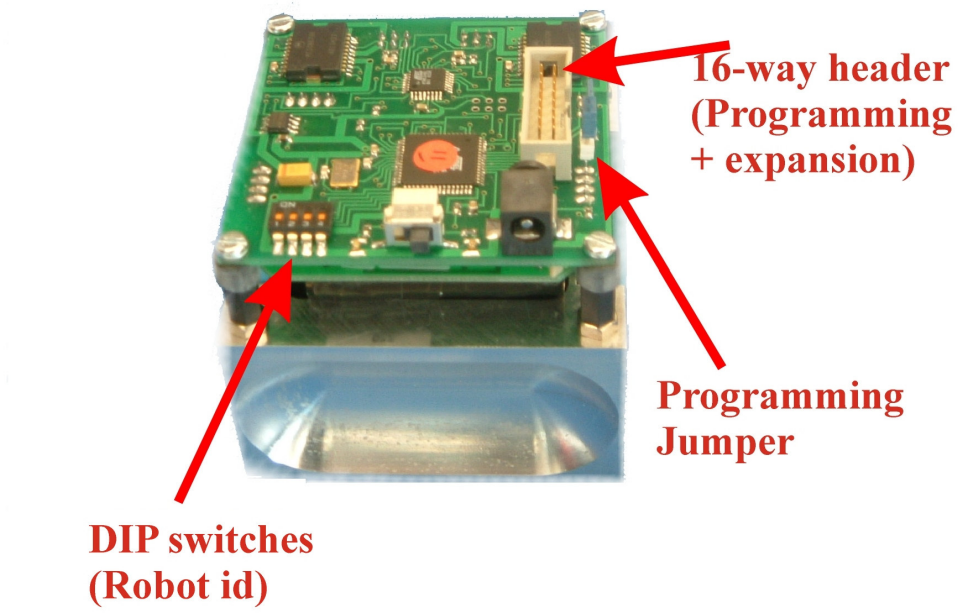
The switch and LEDs are accessed via the machined slot.

The two LEDs are under software control, with 330ohm inline resistors provided. With the standard software, the 'active' LED flashes constantly while the robot is running and the 'comms' LED lights up each time a command is received.

The charge socket takes a 2.5mm DC plug with positive inner connection.
This connects direct to the battery, so charging can be carried out with the robot either on or off.

Internal View

The case lid simply pulls off and refits on rubber friction stops.
The inside looks like this –



The programming cable provided can be plugged into the expansion port, connecting the robot to a PC parallel port for rewriting the robot firmware.

When programming, the 2-pin jumper should be removed

NOTE: removing the jumper avoids possible damage to the radio board during programming

(It is important to be careful about this, as programming usually still works with the jumper fitted.)

The 4-way DIL switch is connected to micro pins and can be read under software control.
This can be used to distinguish individual robots in a small group.

Drive Chain

The motors are driven by 6 x 1.2v (AA) cells through a low-resistance driver I.C. with a slow-acting current limit at about 5A. Maximum speed of an unloaded motor is in the region of 6-8000 rpm.

Position Encoders

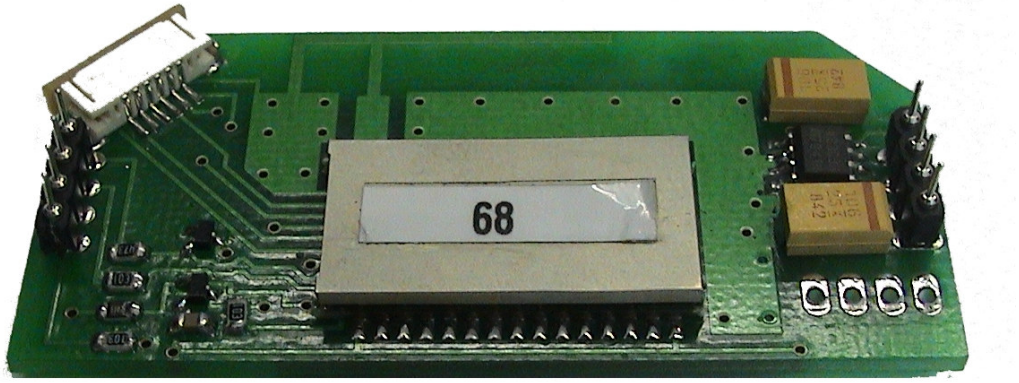
The motor shafts drive the wheels through an 8:1 gearing. The motors incorporate quadrature encoders giving 512 position-pulses per rotation. The wheels are 52mm in diameter, so one encoder pulse corresponds to just under 0.04mm of movement.

Batteries

Each robot contains two 3×AA-cell battery packs (nominal 1.2v per cell, 1300mAh). The robot is supplied with a NIMH fast-charger that can charge these completely in about 1-2hrs. Batteries will last about 1hr typical continuous use (much more if not moving!). Higher-capacity batteries are also available in the same physical size.

Communications Board

A Bluetooth communications card is incorporated within each robot. This enables the host PC to communicate with the robot by converting the Bluetooth link to logic-level serial signals connecting to the main board processor –



A PC Bluetooth dongle is supplied that plugs into the USB port on the PC.
This can support wireless links with up to 7 robots at once.

Expansion Port

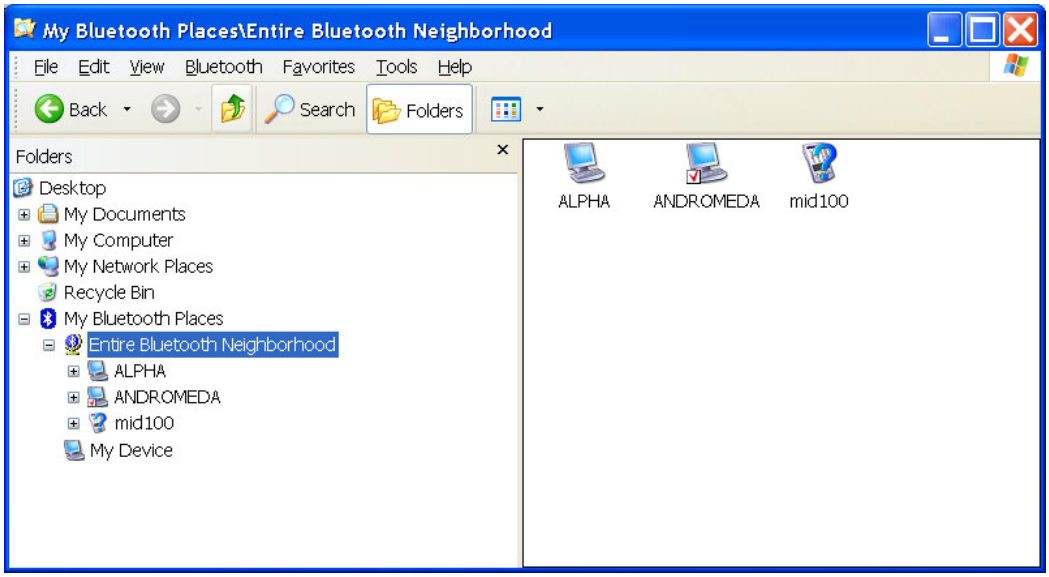
This is used for development programming and for connecting to external devices such as sensors. The connector takes a 16-way IDC plug or board-stacking connector. Details are given in Appendix B.

Bluetooth Communications

Follow the manufacturers instructions (included) to install the PC dongle.

Once installed, double click on the Bluetooth icon (bottom right hand corner). This will bring up an explorer window showing “My Bluetooth Places”.
Make the ‘folders’ view visible, and click on “Entire Bluetooth Neighbourhood”.
Then select “Bluetooth\Search for Devices”, and wait while the machine scans for available contacts.

If the robot is turned on and in range you should get a window looking somewhat like this –



The robot is the device called “*mid100*”: MIABOT communications modules are all identified as “mid<X>” , where <X> is a serial number.

N.B. The ‘mid’ number of each robot is shown on the underside.

To link to the robot, first right click on the icon and select “Pair Device”.
At this point you will be prompted to enter a password, by clicking on the blue tray icon (bottom right): Key in the pass code “1234”.
Right click again and select “Discover Available Services”, which should after a short wait come back with something like “SPP slave” or “Bluetooth Serial Port”.
Right click on the icon once more, and select “Connect to Bluetooth Serial Port” (or similar), and the robot is connected to a virtual COM port.

The identity of the COM port now appears in the status bar at the bottom of the explorer window, or can be found from right-click, “Properties” on the icon.
The device icon is highlighted (usually green) when connected.

(N.B. The actual COMport number will depend on the order of connecting, and other devices in your system).

NOTE: this is a ‘safe’ connection method: The process can be made simpler once you are confident of the details. (Unfortunately, the precise operation depends on the version of the Bluetooth tray software and the operating system in use).

Verifying Robot Communication

It is a good idea to confirm that communications have been established by using a terminal program such as Hyperterminal.

Set up a new connection to connect to the appropriate COM port (e.g. COM4): Most of the ordinary serial settings have no effect, but ensure that “no handshaking” is selected.
Within the Hyperterminal “Settings” tab, “ASCII setup” page, it is best to set “Send line ends with line feeds” and “Echo typed characters locally”.

Connect, and type a command like “[t]”, and the robot should then respond with e.g. “<test#00>”.
This command response contains the robot ID, set by the switches at the back of the robot.

You can try various other commands by referring to the available commands listed in the “Example Code Command Protocol” section, below.

Bluetooth Links

Each Bluetooth link is a dedicated, secure two-way channel established exclusively between the two devices (the ‘pairing’).
It appears to PC applications programs as a “virtual COMport”, which can be connected to much like an ordinary serial port. At the robot end, it appears as logic-level serial signals.

The PC dongle acts a Bluetooth ‘master’ device (which can establish links) while each robot is a separate ‘slave’ device. A slave device can only be paired with one master at any one time.

If radio contact is lost, the link will be automatically restored when it is regained. However, whenever the robot or computer is powered off, the link must generally be re-established, including re-typing the password.

Note on Multiple Links
To connect to more than one robot you may need to configure extra virtual serial ports. The steps required vary, depending upon the Bluetooth host software installed, but here is an example :-
- Right-click on the Bluetooth tray icon, select setup/configuration.
- Select “Client Applications”. At the bottom of the panel should be a button “Add COM port”: After selecting this, enter a name for the connection e.g. “MIABOT1” and select an available COM port.
- Repeat this for each additional COM port required.

Standard Command Protocol

These are the commands supported by the standard robot control software :–
(from v2.1)

All commands begin with the command-start character '[', and end with the command-end character ']'. The first character after '[' identifies the command, after that bytes are free-format, determined by the specific command used.
Extra characters after command-arguments and before ']' are ignored and extra characters between commands (after ']' and before '[') are also ignored.

When the processor starts up or reboots, the robot emits the sign-on message e.g. "<Merlin Systems Corp. Ltd : Miabot Pro OS 2.1 >".

Simple commands

`[s]` - "stop"
sets both wheel speeds to 0

`[t]` - "test"
test communications: Returns a string containing the robot-id, e.g. "<test#03>" for robot 3.
The i.d. is set from 0-15 using the 4 DIL switches.
If id=15 the robot runs the default test sequence on powerup.

`[?]` - "version"
Returns the powerup message, including the firmware version
-e.g. "<Merlin Systems Corp. Ltd : Miabot Pro OS 2.1 >"

Parameter control

Various control functions use stored 'parameter' values.
All values are integers, with a potential range of ±(2^31 ≈ 1 billion).

They are accessed via the **`[.]`** command, which has a number of different forms :–

`[.<name>]` - "read param"
E.G. "[.xM]" , which might return " xM = +00000007".

`[.<name>=<value>]` - "write param"
E.G. "[.xT=-503]", might return " xT = -00000503"

`[.=]` - "reset params"
All parameter values are reset to the factory defaults

`[.]` - "list params"
This lists all params in order, e.g.

```
  rT = +0000010
  rI = +0000010
. . .
```

After each line, the ‘list-all’ command waits to receive a confirmation character before showing the next parameter setting.
If a ‘.’ character is sent, the rest of the params are shown at once.
If a ‘/’ character is sent, the rest of the params are skipped.
If anything else, the next param is shown.

NOTE: the speed control code does not operate while a list-all command is in progress, so this should not be used when the robot is moving.

Speed control

Wheel speeds are specified as a number with a fixed scaling, from 0 to approximately 2000 maximum (positive or negative).
The actual rate in terms of pulses-per-second is speed*50, so that a speed of 1000 is actually 50,000 pulses per second, i.e. a linear speed of approximately 2.0m/sec.
(In practice, speeds of up to 1000 are readily achieved, even when batteries are running low.)

[=<##l>,<##r>] - "set speed decimal"
Set wheel speed (decimal)
<##l>is decimal number, controlling the left wheel set speed
<##r>is decimal number, controlling the right wheel set speed

[-<#l><#r>] - "set speed byte"
Set wheel speed (binary)
<#l>is a single binary byte, controlling left wheel set speed
<#r>is a single binary byte, controlling right wheel set speed

These values are scaled to actual wheel speeds (see above) by multiplying by 8.
(The actual multiplier is a parameter “bV”, which can be changed).

Stepwise movement

[<] - "turn left"
[>] - "turn right"
[^] - "step forward"
[v] - "step backward"

rotate or move forward/backward by set distances.
As for speed commands, the wheels are controlled independently.
The movement speeds and distances are fixed values, set by separate commands (see below).

[d<?><##>] - "set step distance"
Control step-distance to move (forward/backward) or turn (left/right) for all 'step' commands

<?> is a movement-type character: '<' or '>' to set turn-distance, '^' or 'v' to set movement-distance
<##> is a decimal number, setting the number of encoder steps. This has a range of up to (2^31). N.B. this should always be positive! In addition, a 0 means 'forever'.

E.G. "**[dv1730]**" = set linear movement distance to 1730.

The move ('v' or '^') setting is stored independently of the turn ('<' or '>') setting. Standard (reboot) settings are 4000 for movement, and 667 for turn.

[x<?><##>] - "set movement rate"
Control set speed for all movement or turn operations.
<?> is a movement-type character: '<' or '>' to set turn-rate, '^' or 'v' to set movement-rate
<##> is a decimal number, read as a speed setting

E.G. "**[x<110]**" = set turn rate to 110.

The move ('v' or '^') setting is stored independently of the turn ('<' or '>') setting. Standard (reboot) settings are 100 for move, and 50 for turn.

Distance-controlled commands

Turn at set turn-rate, or move forward/backward at set movement-rate (see 'x' command, below), for a specified distance.

[m<#>] - "left by"
[n<#>] - "right by"
<#> is a single binary byte, 0-255, controlling the distance. 0 means 'forever'.
The speed is the 'turn rate' ('[x<' or '[x>') command setting, described above.
The actual distance is the byte-value multiplied by 25, for a maximum of a somewhat more than a whole turn.
(The actual multiplier is a parameter "b<", which can be changed).

[o<#>] - "forward by"
[p<#>] - "backward by"
<#> is a single binary byte, 0-255, controlling the distance. 0 means 'forever'.
The speed is the 'movement rate' ('[xv' or '[x^') command setting, described above.
The actual distance is the byte-value multiplied by 25, for a maximum of a somewhat more than a whole turn.
(The actual multiplier is a parameter "b^", which can be changed).

Acceleration and Deceleration controls

These are controlled by special parameter settings :–

[.rT] - "ramp time period"
[.rI] - "ramp increment rate"

These two control maximum acceleration rates by limiting the rate of change of the programmed speed to ‘rI’ speed units per ‘rT’ milliseconds. This applies to both acceleration and deceleration.
Standard (reboot) settings are rT=10, rI=10.

[.xS] - "distance-speed scaling"
[.xT] - "distance threshold"
These two control the deceleration to stop at a fixed position (for a distance-based command).
Speed during a distance-command is limited to ‘xS’ times SQRT(distance).
Speed is 0 (and distance operation terminates) when position-error < ‘xT’.
Standard (reboot) settings are xS=5, xT=10.
E.G. - at 100 encoder-pulses distance (about 4cm), max speed will be 5*sqrt(100)=50,
- movement will stop within 10 steps (about 0.4mm) of required endpoint.

Sequence controls

The robot has a “current test sequence” of stored commands that can be rewritten or executed. When executing –
The two LEDs flash alternately at a rapid rate.
Commands from the stored sequence are executed in turn.
Execution stops at the sequence end, or when a new serial command is received.
The robot is stopped when the sequence ends or is aborted.
When performing movement commands, the sequence waits for any set distances to be completed before executing the next command.
The 'wait time command' (see below) can also be used to pause the sequence for a fixed time (while robot runs on at a set speed).
The sequence is also run on reboot, if the robot id is set to 15 (all switches ON)

[~] - "do sequence"
Perform the current stored command sequence
[N.B. '~' itself can be added at the end of a sequence, to make it repeat forever]

[$] - "clear sequence"
Erase the stored sequence

[+<command>] - "add sequence command"
Add a command to the stored sequence
<command> is any other ordinary command (minus the usual square brackets)

[w<##>] - "wait time"
Used only within sequences, to pause sequence operation for a set time.
<##> is a decimal number, specifying the number of milliseconds to pause sequence execution.

E.G. the following commands establish a new sequence of movements:

```
[$]
[+^]
[+w1000]
[+v]
[+w1000]
[+=20,-20]
[+w2000]
[+s]
[+~]
```

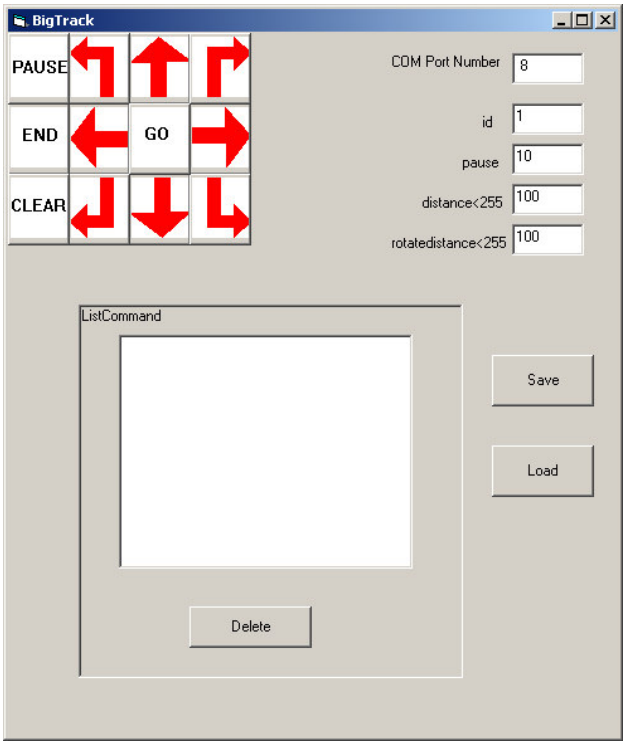
This will cause the robot to move forward, pause 1 sec, move back, pause again, spin slowly for 2 secs, then repeat. The sequence is run by issuing the command [~].

Bigtrack Simulator

Bigtrack simulator is a simple VB application that can be used to drive the Miabot through a series of movements.

The program is found on the installation disk in “BigtrackBT\bigtrackBT.exe”.

First connect to the robot over the Bluetooth link, as described above in the “Bluetooth Communications” section.
Now run the program, which puts up a single control window :–



Type the number of your virtual COM port into the “COM Port Number” box.

You can now use the red arrow keys to make a series of required movements.

The edit boxes on the top right can be used to alter the distance associated with each movement, and the “pause” box adjusts the delay given between successive movements.

As each movement is selected it is written to the listbox below.
The whole sequence can then be repeated by hitting the “GO” button.

Joystick Control

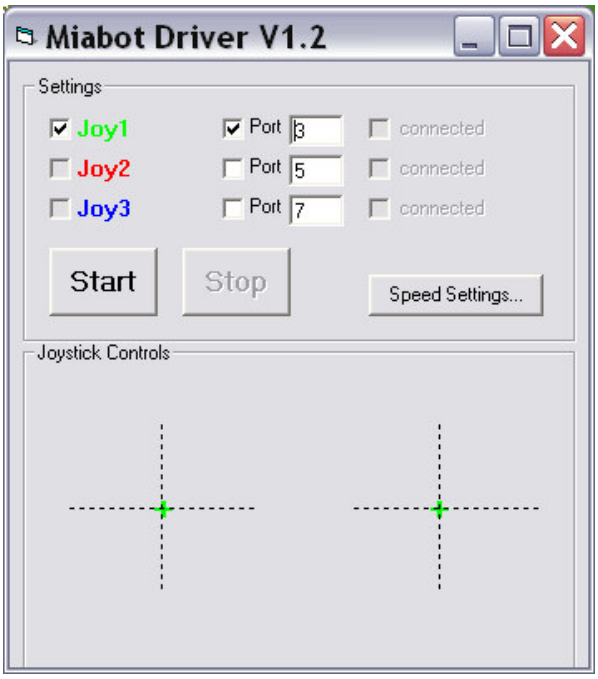
The joystick control software is an interactive application which allows robots to be controlled by joypads attached to a PC. It can drive up to three robots at once. This can be used for simple robot testing, demonstrations and interactive robot football competitions.

Installation

Run the program 'Miabot Joystick Driver\SETUP.EXE' from the installation disk. This installs the program; you can then run it from your Programs menu.

Operation

First connect all the robots to virtual COM ports via the bluetooth link, as described above in the "Bluetooth Communications" section.
Now run the application, which presents a single control window :–



The tick-boxes at top-left show which joysticks have been found (in the example shown, just one). All available joysticks are ticked (enabled), but you can disable unwanted ones.

The tick-boxes in the centre and the adjacent "Port" boxes control which COM ports will be used to control a robot from each joystick.
If any of these are un-ticked, the program will not connect to this COM port (i.e. robot). This can be used for testing joysticks.

Hit the "Start" button to start control over the enabled COM ports.
When activated, the grey "connected" boxes will show ticks for ports that have been successfully connected.

The “Joystick Controls” panel below shows the movement of the left + right analogue joysticks, in different colours (Green, Red, Blue) for each of three possible devices.

NOTE: The program only works with dual analogue joypads, which must be installed and calibrated using the standard Windows facilities before use.

In-Use Controls

Robot speed is controlled by forward and backward movement of the left-hand stick. Steering is controlled by left and right movement of the right-hand stick.

The “Stop” button stops all the robots and disconnects from the COM ports.

Pressing the “Speed Settings” button brings up a control panel :–

These sliders control the maximum speed and rate-of-turn applied to the robot (for full stick deflections), while the “accel” sliders adjust the speed of response.

The default values are set for reasonable performance in robot-soccer type demonstrations. They can usefully be reduced for smoother movements in other applications, e.g. maze tracking.

Robot Firmware Development

Overview

MIABOTs are supplied with standard software already installed.
The 'c' source code for the example software is supplied in the files PROBOT.C, PROBOT.H, VARDEFS.H and MAKEFILE on the "Development Kit" disk.
As supplied, these are configured for the WinAVR development system.
When correctly programmed, the example software will flash the LEDs when the robot is turned on. It also implements the demo sequence, and most of the standard commands detailed above.

Compiler Installation

WinAVR is a freeware development environment containing a port of the well-known GCC compiler. WinAVR is supplied as a self-extracting installer: Run this to install the software.

Programming

The robot is in-system programmable and new firmware can be downloaded into the robot via the programming lead.

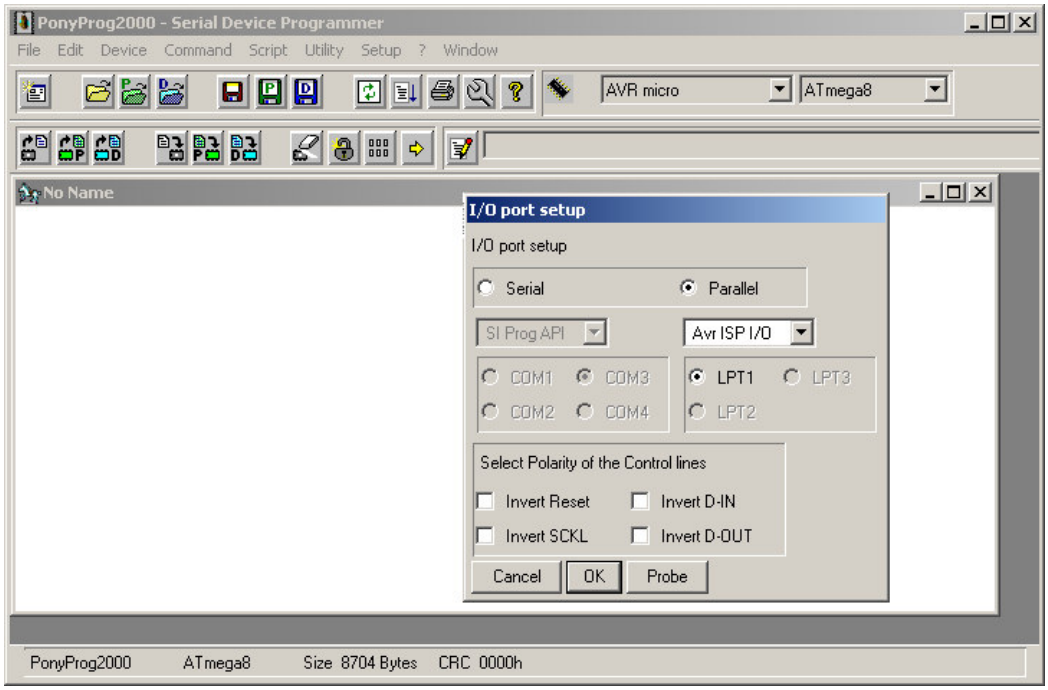
We supply a freeware development environment called WinAVR. This includes the GCC 'c' compiler for Atmel AVR micros, a full 'c' library, a programmer's editor, and download and debugging utilities.

At present, we recommend starting with the "PonyProg" programmer instead of the download tool supplied as part of WinAVR.
PonyProg is a freeware utility supplied by www.lancos.com, which can be used to take an Intel format hex file (output from GCC) and download it into the robot.
We have supplied the latest version of PonyProg as a zipped file. Please unzip the setup.exe and run.

At this point, remove the two-pin jumper on the top of the robot PCB beside the expansion port. This is important to protect the radio board from the programming process (see 'Internal View', page 7).

Connect the programming cable between the robot and the PC parallel port, and turn the robot ON.

Now run up PonyProg :–



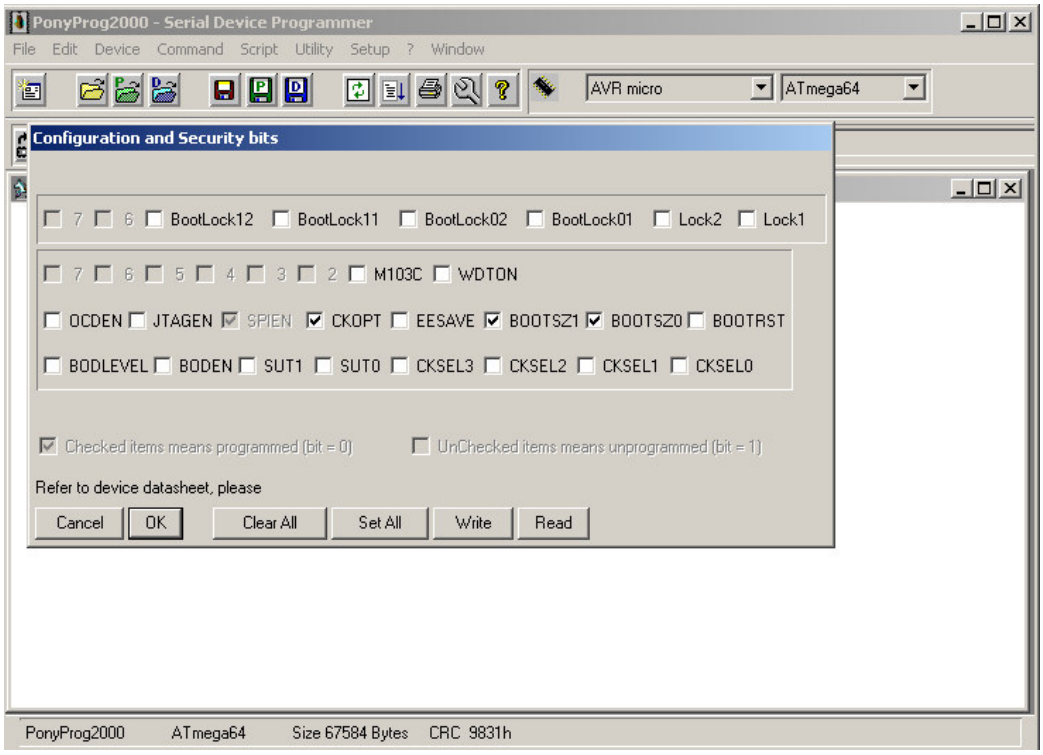
The first time you use PonyProg, you will need to configure it, selecting the setup options shown above:

- First select "Interface Setup" from the Setup menu. Select "Parallel" for the programming port, "LPT1" button for the port (or whichever parallel programming port you wish to use), and "AVR-ISP I/O" (the hardware interface type) from the dropdown.

- Next, run "Calibration" from the Setup menu: Follow the instructions to calibrate PonyProg for the machine speed (***N.B. this requires the robot to be connected***).

- On the main toolbar, select "AVRMicro" as the processor family and "ATmega64" as the processor type.
PonyProg should now be set up correctly.

To check correct communication with the programmer, you can now read back some information from the robot:
Select “Command\Security and configuration bits”, which brings up the control pane for the Mega64 fuse and lock-bit settings.
Turn all the check-boxes off, then hit the ‘Read’ button to refresh the settings from the robot.
It should look like this :-



The robots are shipped with these Fuse settings already configured. Do not change anything (or hit “Write”) unless you are quite sure (!).

(N.B. The processor clock controls, CKSEL0..3, are set for the high-speed external crystal which is 1111: Refer to Atmel datasheet, and note that these bits read in reverse logic in PonyProg.
With this setting, it is very important that the CKOPT fuse is also enabled, or the clock may fail, making it impossible to program the robot any more !).

If you are unable to read the fuse settings, some common causes of failure are :–

i) Cable is not plugged in.
ii) Robot is not turned on.
iii) Calibration timing is not correct (run calibration from the setup menu).
iv) Some PC’s are not always compatible, try downloading an older version of PonyProg and retry.

Another simple way of verifying the basic connection is to select “Command\Reset”: The LEDs should flash as the robot reboots.

Once communication is established, you can try reprogramming the example program:–

First save a copy of the existing contents:
- Select "Command\Read Program (FLASH)". The main window should fill up with code read from the robot.
- Select "File\Save Program (FLASH) File as…", and save the result as a .HEX file, e.g. "Miabot Original.hex".

Now, load + program the new file:
- Select "Command\Erase". This wipes the whole chip. You can confirm that the light no longer flash when the robot is turned off and on.
- Now select "File\Open Program (FLASH) File …", and load the example code "PROBOT.HEX".
- Select "Command\Write Program (FLASH)", and PonyProg will reprogram the chip and verify the data. The robot should work normally again.

Alternative Programming Methods

PonyProg limitations

Current versions of PonyProg have some known problems:

1. For reliable results it may be necessary to always manually erase the device (Command\Erase) prior to programming: PonyProg has a configurable automated programming sequence, which can do this for you automatically, but we have found that using this causes the verify cycle after programming to fail intermittently.
2. Reading Oscillator Calibration bytes sometimes seems to corrupt the program window data, leading to bad program data. So always re-load after using these functions.
3. Fuse settings cannot be saved, verified or programmed from a file.

WinAVR / AvrDude

For more advanced use, we recommend the AvrDude programmer application that comes with WinAVR.
This is a command-line utility rather than GUI based, but can prove more reliable than PonyProg and can also solve some PC compatibility problems.

The programming lead provided can be used by selecting the "stk200" parallel programmer type for AvrDude operations.
For example, the command "`avrdude -pm64 -cstk200 -Plpt1`" verifies the programmer connection, reporting either the device signature or an error.

See AvrDude documentation for full details.

Atmel AVR Studio

Unfortunately, it is not currently possible to program from AVR Studio via a parallel lead. Instead you must buy a serially-controlled programmer such as Atmel's AvrISP. See Atmel website for details.

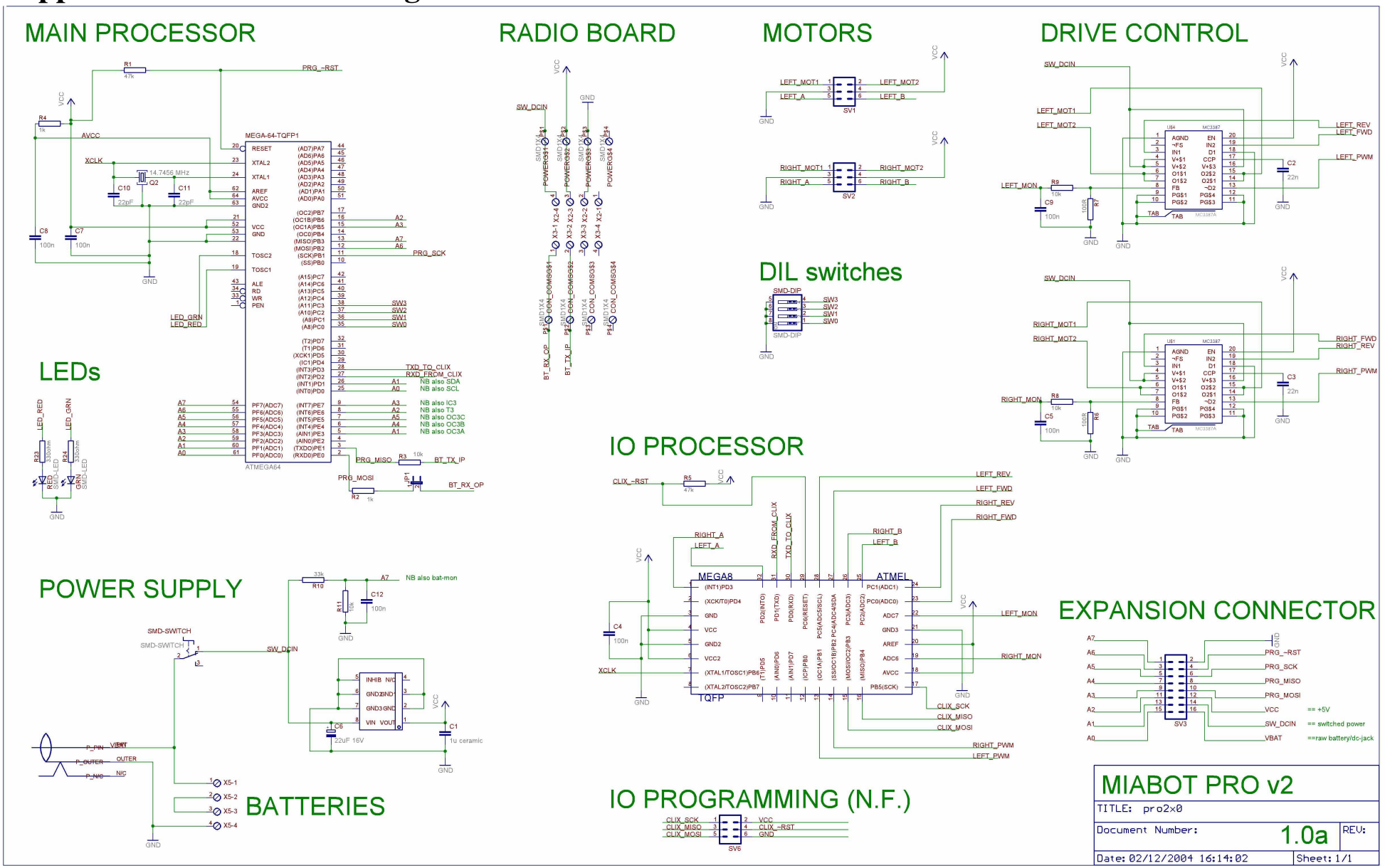
Writing Robot Programs

You can now modify the example program as you like, or write your own code. The output from the GCC compiler should be a .HEX file – see GCC documentation and the example MAKEFILE provided.

NOTE: WinAVR can also interface to Atmel's "AvrStudio" free development environment. In particular you can *simulate* code in AvrStudio by exporting it as a .ELF file. With suitable hardware you can also use an in-circuit emulator.

A good starting point is to visit the WinAVR homepage at winavr.sourceforge.net. In the 'Documentation' section, you can find the file "install_config_WinAVR.pdf", which gives a concise and readable introduction.

Appendix A : Schematic Diagram



Appendix B : Expansion Port Signals

These are the connections on the main 16-way IDC header –

signal	pin	processor pins	**an/dig**	**ints**	**i2c**	**spi**	**pwm**	(other)
A7	1	PF7 PB3=miso	a7	.	.	miso	.	**(battery)**
A6	3	PF6 PB2=mosi	a6	.	.	mosi	.	.
A5	5	PF5 PE5=int5/oc3c	a5	int5	.	.	oc3c	.
A4	7	PF4 PE4=int4/.oc3b	a4	int4	.	.	oc3b	.
A3	9	PF3 PE7=int7/ic3	a3	int7	.	.	oc1a	ic3
A2	11	PF2 PE6=int6/t3	a2	int6	.	.	oc1b	t3
A1	13	PF1 PD1=int1/sda	a1	int1	sda	.	oc3a	.
A0	15	PF0 PD0=int0/scl	a0	int0	scl	.	.	.
						.		
GND	2					.		
¬RESET	4	RESET				.		(low=reset)
SCK	6	PB1=sck				sck		
PDO/MISO	8	PE1						
PDI/MOSI	10	PE0						
VCC	12							+5V rail
SW_DCIN	14							switched charge/battery voltage
VBAT	16							unswitched charge/battery voltage

(The signal names relate to the system schematic - see Appendix A).

The main expansion signals, **A0..7**, are carried down one side of the 16-way connector (odd numbered pins).
All eight of these signals can be configured to provide a simple digital input or output, or analogue input capability. Each pin also has a built-in programmable pull-up resistor.

Various special uses of the expansion-connected signals are also shown, as follows –

an/dig = normal digital i/o or analogue adc input (PORTF)

ints = interrupt capable inputs

i2c = "two-wire" serial interface (suitable for I^2C bus)

spi = synchronous serial interface (spi, microwire etc.)

pwm = pulsewidth modulated output

See processor datasheet for details of i/o pins capabilities.

SPECIAL WARNING NOTE - signal A7

Signal A7 is ***also*** connected to a potential divider (33k/10k) to monitor the robot battery voltage.
This can still be used as a general-purpose adc/digital-io line, as long as the connected signal has a low enough source impedance.
However, this will prevent battery monitoring, so in practice it is usually better to avoid using this signal.

Appendix C : Io Processor Control Protocol

The Miabot Pro uses a subsidiary "io processor" (Atmel Mega8) to handle the wheels. The io processor counts the motor encoder pulses, and generates PWM signals to drive the motors.

Both processors run off the same 14.7456MHz crystal clock, and communicate via an asynchronous serial protocol running at 14.7456Mhz/32 ~=460kBits/sec
(i.e. UART UBRR value = 1)

The protocol is a simple master-slave command set, where each command starts with a single distinguishing character sent by the master.
At present (v2.0), there are only four commands :–

command	character	description
RESET	0	set current position counters to 0
STOP	–	set wheel drive-levels to zero
QUERY	?	read wheel positions
SET	=	set wheel drive-levels

The wheel position counts are 16-bit values, which wraparound on overflow.
Motor drive levels are 8-bit signed values (i.e. -127..+127, and ±128 same as 0).

Operation of the RESET and STOP commands is immediate, but the other two commands use extra 'handshaking' bytes to synchronise the exchange of multiple databytes, as discussed below...

QUERY command

This samples both current wheel positions, and responds with a four-byte sequence containing the wheel positions.
The io processor waits to receive a confirm character ('.') after sending each data byte, before it can send the next.
The master therefore sends a confirm character after receiving each data byte –

send '**?**'	→	io outputs first byte (left-wheel HIGH)
send '**.**'	→	io outputs second byte (left-wheel LOW)
send '**.**'	→	io outputs third byte (right-wheel HIGH)
send '**.**'	→	io outputs fourth byte (right-wheel LOW)
(begin next command)		

If anything other than a '.' (hex 2E) confirm character is sent, the command aborts prematurely.

SET command

This accepts two new drive-level data bytes and then updates the output drive levels.
The io processor sends a confirm character ('.') after receiving each command byte.
The master waits for the confirm characters before sending the next data-byte.

send '**=**'	→	io outputs confirm byte '.'
send right-hand drive-level byte	→	io outputs confirm byte '.'
send left-hand drive-level byte	→	io outputs confirm byte '.'
(begin next command)		