PICBOT PERIPHERALS

This document describes peripherals connected to the PicBot PCB, as well as the connections themselves. Peripherals are anything that is connected to PICBot for the purpose of input or output, such as the SCI, SPI, sensors, servos, motors, etc.

Some of this is informational, for further development; some is to describe peripherals already in use with PICBot; while others are possible configurations gleaned from research that may one day be found handy.

Table of Contents

Table of Contents....1
A/D Converters....2
Result Register Setup....2
Hardware Notes....2
Motors....3
Unipolar Stepping Motors....3
Wiring diagram:....3
Coil Control:....3
Stepping Motor Speed Control....4
Theory....4
PIC Design....4
Future Considerations....4
PC Connections....5
Modem-to-Modem....5
Theory....5
Line Simulator – Schematic....5
PC Port Pins....5
DB9/DB25 Pin Layout....5
PC Port Signals....5
DB9....5
DB25....5
PC Parallel Port to LVP (2-Bit)....7
PC Serial Port to SCI....7
Cable Layout....7
Servos....8
Identification....8
How to Use....8
Red Wire....8
Black Wire....8
White Wire....8
SCI Port Sharing....9
Theory....9
SPI Port....10
Current (PicBot - 1)....10
Connector....10
Proposed (PicBot - 2)....10
Multiple SPI Ports....10
The Selection Alternative....10
The Daisy-Chain Alternative....11
Wiring -- Master and Slaves:....11
Operation: (Rules of Protocol)....11
Connectivity Alternatives....12
Single-Slave....12
PicBot - 1 Compatible....12

A/D Converters

Result Register Setup

Since I and seemingly everyone else are having problems getting an appropriate result from the A/D conversion result registers, I decided to go with the “Left-Justified” approach, in which the result is left-justified in the higher register and the lower two bits are ignored. Eight bits should be sufficient to get a good range of results.

The problem I had specifically is that the 2-byte result is that the lower byte always equaled the higher one. When converting from three nibbles (ignoring the top nibble, higher byte) to a 3-byte hex value, the only results I ever got was “101”, “202” and “303”. There is an off-chance this is a bug in my own code, but I doubt it after reading of other’s distress in this area.

If you decide to go back to a right justified two-byte, ten-bit result, set the ADCON1:ADFM to 1 and try the code in the SendAD subroutine:

```
...
movfw   ADRESH
call    TxPutHex
movfw   ADRESL
call    TxPutHex
call    TxPutCRLF
...
```

This will output a 4-byte hex output, the first byte always being zero.

Hardware Notes

The PIC Reference Manual recommends a maximum impedance on an AD input pin of 10Kohms, otherwise it will degrade the conversion process.

Motors

UNIPOLAR STEPPING MOTORS

Such as the M82101, a small unipolar stepping motor with 7.5 degree step.
(See Jamesco catalog and resources PDF for other specifications).

Wiring diagram:

Wire	Coil
----	----
Orange	1
Red	Common
Yellow	2
Brown	3
Green	Common
Black	4

Coil Control:

- Single step, single coil (less power use, less torque):

```
1000
0100
0010
0001
1000
...
```

- Single step, two-coil (more power, more torque):

```
1100
0110
0011
1001
1100
...
```

- Half-step (more postitions, compromise on power):

```
1000
1100
0100
0110
0010
0011
0001
1001
1000
...
```

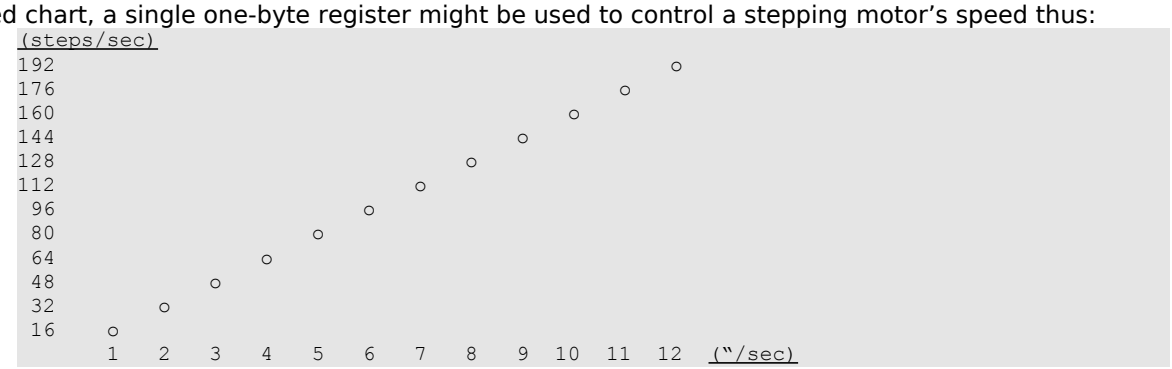
STEPPING MOTOR SPEED CONTROL

Theory

A typical stepping motor has 7.5 degree steps.
Given a theoretical 3" circumference wheel:

- ➔ One revolution = 3" of travel.
- ➔ Each revolution = 48 steps.
- ➔ 16 steps/second = 1"/second speed.

A speed chart, a single one-byte register might be used to control a stepping motor's speed thus:



PIC Design

The simplest means to store stepping motor state, for two motors, is in another single one-byte register:

```
Register Bit
        7__     Motor1 Direction (1=Forward, 0=Reverse)
        6
        5       Motor1 Speed (0-7)
        4__
        3__     Motor0 Direction (1=Forward, 0=Reverse)
        2
        1       Motor1 Speed (0-7)
        0__
```

Note, to indicate that a motor (or both motors) are not moving, set speed to 0.

- ➔ To indicate that one or both motors have been deactivated (thus freeing a port for other use), simply set both direction and speed to 0.
- ➔ The activation state of the motors can easily be checked by the PIC program as a port activation switch.
- ➔ This register can be set easily by passing the entire byte as a command from the host (DosBot) system.

Since there are only 8 speed values handy, a more logarythmic speed variation might be more useful, the PIC converting the 3-bit value to a delay between steps, as such:

Speed	Millisecond Delay	=	Steps/Sec	=	Dist. Travelled
0 (Stop)	N/A		0		0"
1 (Slow)	255		4		¼"
2	127		8		½"
3	63		16		1"
4	31		32		2"
5	15		64		4"
6	7		128		8"
7 (Fast)	3		256		16"

Future Considerations

- ➔ Obviously some calibration will be needed, based on wheel size and other real factors.
- ➔ Since stepping motors, unlike servos, do not have an automatic "ramp-up" or "ramp-down" capability, some elegant programming of delay use may be necessary to ward off the "herky-jerky" operation that is liable to occur with the above design.
  - ➔ Simply setting a target speed and stepping through the set speeds one at a time until reaching that speed may suffice as a ramp-up/down measure.
  - ➔ Stepping through longer delays to shorter ones would provide a natural accelleration, although one of each may be too fast to do any good (or may be perfect...).

PC Connections

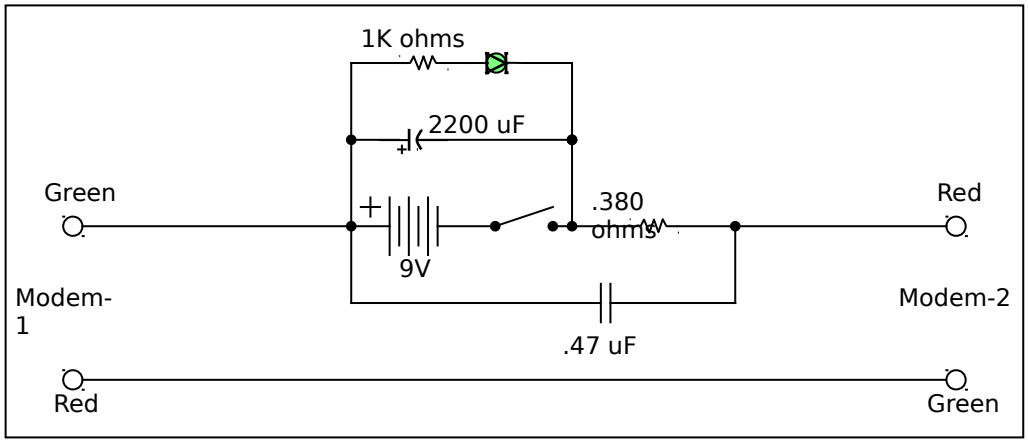
MODEM-TO-MODEM

Theory

In preparation for going “wireless”, this is used to test the ability of one PC to communicate (read: telemetry) with another “onboard” PC (a stripped-down laptop) using two modems without the telephone company being between them. This assumes that the only external serial port on the Bot side is being used to interface to the PicBot subsystem.

Since two modems can normally not communicate with each other directly, since they require the current of a “live” line between them, this circuit was devised to simulate that line:

Line Simulator – Schematic



PC PORT PINS

DB9/DB25 Pin Layout

```
        DB9 (Female)                    DB25 (Female)
         5       1                       13                          1
         o o o o o                       o o o o o o o o o o o o o
          o o o o                         o o o o o o o o o o o o
          9     6                         25                         14

        DB9 (Male)                      DB25 (Male)
         1       5                       1                          13
         o o o o o                       o o o o o o o o o o o o o
          o o o o                         o o o o o o o o o o o o
          6     9                         14                         25
```

PC PORT SIGNALS

DB9

Pin	Color*	Signal	Direction
1		CD	
2		RD	In
3		TD	Out
4		DTR	
5		SG	-
6		DSR	
7		RTS	
8		CTS	
9			

DB25

Pin	Color*	Signal	Direction
1	White/Tan	_STROBE	Out

```
2       White/Brown    DATA 0         Out
3       White/Violet   DATA 1         Out
4       White/Orange   DATA 2         Out
5       White/Yellow   DATA 3         Out
6       White/Green    DATA 4         Out
7       White/Blue     DATA 5         Out
8       White/Purple   DATA 6         Out
9       White/Gray     DATA 7         Out
10      Brown/Brown    _ACK           In
11      Brown/Violet   BUSY           In
12      Brown/Orange   PAPER END      In
13      Brown/Tan      SELECT         Out
14      Brown/White    _AUTOFEED      Out
15      Black/White    _ERROR         In
16      Pink/White     _INIT          Out
17      Red/White      SELECT IN      ?
18      Orange/White   GROUND         -
19      Green/White    GROUND         -
20      Blue/White     GROUND         -
21      Brown/White    GROUND         -
22      Gray/White     GROUND         -
23      Purple/Tan     GROUND         -
24      Orange/Tan     GROUND         -
25      Red/Tan        GROUND         -
--      (Bare)         CABLE GROUND   -
```

* Colors may be arbitrtary here, but are shown as (stripe) on (solid).
_ = Active Low signal. Basic, ECP, and EEP have some bidirectional signals.

PC PARALLEL PORT TO LVP (2-BIT)

```
Signal          DB25                            LVP     Signal

_ACK            10 -------------------------- 0       PGD (Out)
GROUND          18-25 ======================== 1       Ground
DATA 2          4 -------------------------- 2       PGM
DATA 0          2 -------------------------- 3       PGD (In)
DATA 3          5 -------------------------- 4       RESET/MCLR
DATA 1          3 -------------------------- 5       PGC
```

PC SERIAL PORT TO SCI

Cable Layout

```
 ----PC Port Pins----                    ---PIC Serial Port---
DB25    DB9     Signal                          Signal _______
                                                              /^\
 2       3        TD ---------------------------- RD . . . . . | |
                                                              ===
 3       2        RD ---------------------------- TD . . . . . | |
                                                              ===
 7       5        SG =========(shield)========== SG . . . . . | |
                                                              | |
20       4       DTR --+                                     =====
                       |                                     |   ||
 6       6       DSR --+                                     |   ||
                       |                                     |   ||
 8       1        CD --+                                     |   ||
                                                             \   |/
 4       7       RTS --+
                       |                                   1/8” plug
 5       8       CTS --+
```

Servos

IDENTIFICATION

A servo is a small motorized device with a single shaft, usually geared to travel within a 180 degree range. They operate by receiving a coded signal, pulses that tell it exactly at what angle to position the shart. Usually used in R/C applications, they can also be handy in robotics, given their small size, the fact that they contain their own control circuitry, operate on 5 volts and are extremely powerful for their size. A standard servo such as the Futaba S-148 has 42 oz/inch of torque.

The servo is internally ganged with a potentiometer that senses what angle the shaft is at all times and matches it against the constant pulse signal being input. If at any time the shaft does not match the signal, the motor activates to position it at that angle. The amount of power applied to the motor is proportional to the distance it needs to travel to attain the correct angle, called proportional control, which makes for very smooth operation, without having to input varying signals for run-up and slow-down speeds.

HOW TO USE

Each servo has three wires:

Red Wire
Connect 5 volts here.

Black Wire
Connects to ground.

White Wire
This is the signal wire. Using what is called Pulse Code Modulation (PCM) the servo expects to see a pulse every 20 milliseconds (.02 seconds). The length of the pulse (off-on-off) determines the desired angle of the shaft.

Pulse		Shaft Angle
1.25 ms	=	0 degrees
1.50 ms	=	90 degrees
1.75 ms	=	180 degrees

These values may vary a bit from one manufacturer to another, but the principle applies to all.

SCI Port Sharing

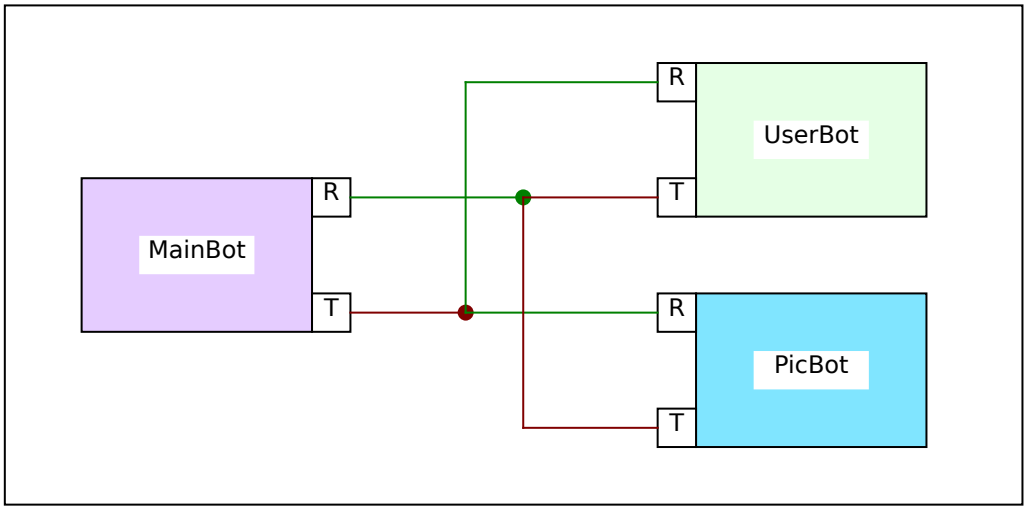
THEORY

The problem: Having only one working serial port for the “Main” PC (the one that will be part of the mobile Bot system).

Since this is the only serial interface, and there are possibly two units to interface, the PIC and the User PC, I can think of only one manner of connecting these three units on a shared serial bus.

Key:

MainBot	The brains of the outfit, a salvaged laptop CPU (and selected peripherals), mounted on the mobile Bot platform running in Basic.
UserBot	For the user to communicate “suggestions” to the autonomous MainBot, this is any fixed PC, running in Basic.
PicBot	The hardware interface, running in PIC assembler.



In this scenario, MainBot is in control of all communication, polling both UserBot and PicBot before either is allowed to respond with any input. In this way, if both of the peripheral modules are responding correctly (that is, returning data or acknowledgments upon receiving a poll), then data collisions should be rare to nonexistent.

UserBot and PicBot should then be assigned unique addresses and should refrain from transmitting any data unless specifically addressed and asked to do so. Likewise, when MainBot transmits data, only the addressed module should pay any attention to it.

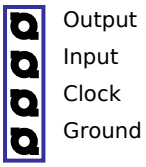
Each input command from the user should be place in an outgoing buffer, input should be inhibited, and UserBot should simply wait to be polled by MainBot. If there is no response in a reasonable period of time, UserBot should indicate that communication is down and provide the user with options accordingly (one of which might be to wait for communication to be reestablished).

PicBot, on the other hand, only outputs data when asked for it, so no buffer is necessary (except perhaps for input from the SPI port). MainBot simply asks for a status report and PicBot gives it what it has at that moment.

SPI Port

CURRENT (PICBOT - 1)

Connector



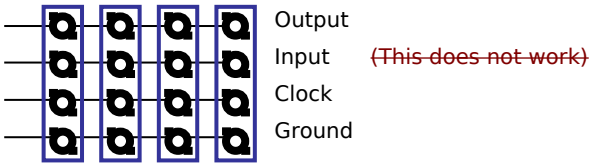
This is fine for a single connection to one sister PCB SPI interface.

PROPOSED (PICBOT - 2)

Multiple SPI Ports

[Note: the following semi-outdated stuff makes a few incorrect assumptions. See the PicBot PCBs (Version 2) document for a description of a very successful means of "sharing" the SPI lines exactly as pictured below. It describes the simple "bus" approach used everywhere.]

Given that the SPI shifts bits into and out of the same register simultaneously, there is no way to "share" SPI ports in parallel as I originally (stupidly) thought, such as how one might share an SCI channel.



[Simply address which slave you wish to hear from and send it an instruction. The selected (addressed) slave responds after the address is received, the master shifting in data until it receives the designated termination signal. All other slaves simply shift out nothing at all—HMM, IS THIS RIGHT? CAN THEY LISTEN TO WHAT IS BEING SAID WITHOUT SHIFTING OUT ZEROES OVER THE OUT LINE WHILE DOING SO???. (Yes, by turning off their output pin until they are selected.)

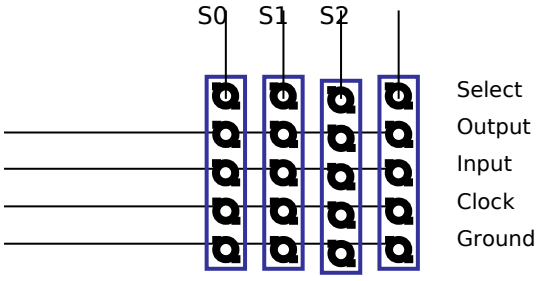
So, everything that follows on this subject is overcomplicated silliness... (I HOPE.)]

The Selection Alternative

More than one sister board would be shifting their stuff overtop of each other in the above scenario, without some kind of selection system, like this:

The down side with this setup is that each selection line (S0-S4) takes up another MCU I/O pin. This particular example takes up a total of seven I/O pins for four SPI ports, where in the impossible scenario above I was trying to do it with the same three pins...

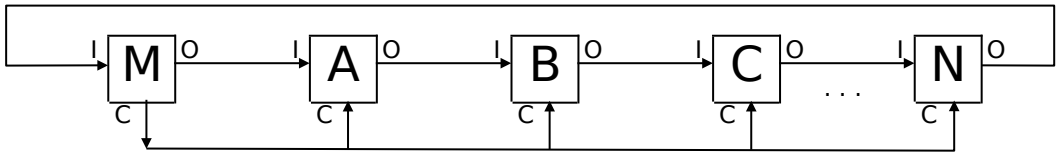
Is there a way to do this while sharing the



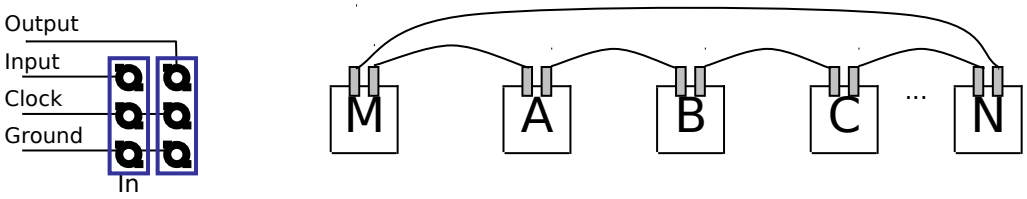
The Daisy-Chain Alternative

At first I thought the following idea was impossible, given the horrendous protocol it would take to pull it off -- until I percolated on it a bit ...

The result? From three I/O pins, as many as seven peripheral units can be communicated without much trouble. Here's a diagram of how it would work physically:



Wiring -- Master and Slaves:



Operation: (Rules of Protocol)

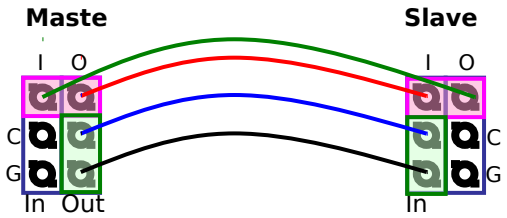
- First, all communication must be split into 4-bit half-bytes (nibbles).
  - The first nibble identifies the destination (when sent by the master) or the source (when sent by a slave). This accounts for the first 3 bits, which identify slaves 1-7, or a dummy data byte of 0.
  - The fourth bit indicates whether this is the first or second nibble of data.
  - The second nibble is a half byte of data.
- The master shifts continuously, since nothing happens without the master making it happen. It is either sending data it has to send, or sending dummy bytes (h'00') so that any bytes shifting out of the slaves can reach its input.
- The master can send data at any time, since none of the data coming in has to be passed on.
- Slaves can only send when the last byte received was a dummy byte (that is, another byte isn't waiting to be passed on).
- Slaves receiving any byte that is not identified as belonging to it simply holds its own data and passes this byte through without doing anything with it.
- Slaves receiving a data byte identified as belonging to it move the data nibble to its input data buffer and eats the byte (does not get passed through).
- Beware: If the master sends a byte to a slave that does not exist, or has been temporarily been unplugged from the loop, will get that byte back as if the byte came from the slave that isn't there. There is no difference in ID between a byte coming from or going to the master.
- Data Streams:
  - Design a buffering system nearly identical to the SCI for holding incoming and outgoing data.
  - Exception: Need separate buffers for each possible SPI slave input (for reconstructing data from nibbles).
  - Since everything is passed as nibbles, sending one whole byte takes two transfers (first nibble is xxx0DDDD, second nibble is xxx1DDDD).
  - The receiver holds the first nibble until the second nibble comes in, then adds it to the receive buffer.
  - Like the SCI, when it receives a CR, it registers a command-received to the chip, which then acts on what is in the buffer.

CONNECTIVITY ALTERNATIVES

Single-Slave

As described above, having two three-pin plugs could be a little bit of overkill if it is to be connected to only one slave, especially when there is no need to connect the clock and ground twice.

For this situation, simply use two two-wire connections, as shown; preferably using color-coded wire to get the input and output connections right:



PicBot - 1 Compatible

This same setup can be used on the current, first prototype PicBot SPI: