Controlling Robobuilder Humanoid robot using Lisp/L#

If you want to give a gripper for your robobuilder, order it now from my Shapeways shop at http://shapeways.com/shop/ip_robotics.

And don't forget to order a servo from Robosavvy to go with it.

Edition 2

Index

Introduction ... 3
Lisp ... 4
Car, Cdr, Cons, List ... 6
So how do these work? ... 6
Function and conditionals ... 8
Controlling the Robot ... 10
More PC Remote mode functions ... 13
Menu driven motions ... 15
Accessing the accelerometer. ... 17
Access the servo directly ... 19
How to play servo moves ... 21
Synchronous moves ... 23
Smooth moves and the In-betweens ... 24
Making complete motions ... 25
Loading data from CSV files ... 27
Controlling the IO on a servo ... 29
Windows based graphics using L# ! ... 30
Plotting and animation ... 32
Real-time accelerometer ... 35
Property lists ... 37
Blockworld / SHRDLU ... 39
Searching and matching ... 43
Adding a gripper with dynamic control of a servo ... 46
A'mazing ... 48
Joystick ... 51
Speech synthesis and recognition! ... 54
Useful features and functions ... 55
Concluding Remarks ... 58
Concluding Remarks ... 58
Appendix A – Function List ... 60
Appendix B – File List ... 63

Introduction

This document is a tutorial guide on programming a Robobuilder[1] series humanoid robot using Lisp based dialect called LSharp.NET [2] (or L#). It will explain the basics of Lisp – although rather briefly – if the reader has no knowledge of Lisp you may well want to look at other books and websites that cover the Lisp language in more detail. This guide will then look at how to control the Robobuilder robot remotely from a PC using .NET. It will show how to control individual servos on the robot, and access its sensors such as the distance sensor and the accelerometer. It then looks at advanced control techniques such as dynamic control of a gripper, using the distance sensor for navigating a maze and accelerometer for auto-balance control. Later chapters deal with interfacing with Windows to provide simple graphics and different input methods such as joystick and speech libraries to control the robot. The final chapters cover a basic introduction to some AI techniques to improve the intelligence of your robot.

This document has been based on use of LSharp.Net (L#), a dialect of Lisp built using .NET framework. Its big advantage is its ability to hook into .NET windows libraries including the standard ones such as "System.Windows.Forms" and also DirectX and Speech (.Net V3) and my own RobobuilderLib.dll for controlling the robot.

All the files required to run the code in the document can be download from the internet and links can be found in Appendix B. A summary of the Lisp/L# functions is Appendix A

Throughout the document are code examples and example sessions. Lisp/L# is an interactive and immediate language and the reader is encouraged to read and try the examples out at the same time. In the examples user input is colour blue and application program output coloured green.

Note: (1) Robobuilder is trademark of Robobuilder co (see http://robobuilder.net for details)
(2) LSharp.Net / L# are Copyright Rob Blackwell

Lisp

Lisp was invented by John McCarthy in 1960 - So this year is its 50th anniversary.

If you're not familiar with Lisp it stands for list processing - this particular lisp (L#) uses a dialect based on "arc" – a description can be found here: http://www.paulgraham.com/arc.html

Here are a few simple examples to get you started. First run the command line environment and get a prompt: ">"

```
C:\> LSharpConsole.exe
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright © Rob Blackwell. All rights reserved.
> (+ 2 3 4 5)
14
```

Assuming you have downloaded and run the console program you should be able to try out the above example. Lisp is a very simple language, it consists entirely of list of thing contained within brackets (). It evaluates the list supplied and returns a result. In the above example (+ 2 3 4 5) is a list that return the result 14. The console program evaluates the list by looking at the first 'atom' in the list (in this case +) and using that as a function to process the remaining elements in the list, in this case, summing the remaining elements. It would also correct to write the following (+ (+ 2 3) (+ 4 5)). Lisp would evaluate the inner elements first to give (+ 5 9) and the sum the outer elements giving 14.

Fundamental things in LISP are atoms and atoms are formed into list by (). So (+ 1 2) represents a list of 3 atoms and if the list is evaluated the + is taken as function and 3 is the result.

Here's another list:

```
> (= x 5)
5
> x
5
```

This shows how to assign a value to the atom "x". If we type "x" to the prompt it evaluates to value of x which in this case is 5. We can now use x in an expression – here using "+" function which adds values:

```
> (+ x x)
10
> x
5
```

Note although (+ x x) added the value of to itself it didn't change its value. Atoms can also be made of ASCII characters - so (a b c) or (a b 1 3) are lists.

I mentioned that Lisp is basically just atoms and list - or s-expressions as they are called. So what is a list and do we work on them? The simplest way is to do a few examples

```
> '(a b c d)
(a b c d)
> '(1 2 3)
(1 2 3)
```

Note the quote (') - this is important! It means don't evaluate the list - take its literal contents. Atoms can be numbers or symbols, and numbers can be integers or floating points (real numbers).

And lists can be me a mixture of elements (and even other lists)

```
> '(a b 1 2 4.0)
(a b 1 2 4.0)
> '( (a b) (c d))
((a b) ( c d))
> (= x '(a b c))
(a b c)
> x
(a b c)
```

But atoms can also be functions - which is where Lisp gets its power - it's a user definable symbolic processing language that can be written in itself using just a few primitives

So (+ 1 2) is just another list:

```
> '(+ 2 3)
(+ 2 3)
> (= x '(+ 2 3))
(+ 2 3)
> x
(+ 2 3)
```

To load files into Lisp/Lsharp you use the function **load**. So with you favourite text editor (vi or notepad) create a file "demo.lisp". Add the following content to the file: (prn "Hello world")

Now start LsharpConsole.exe :

```
> (Load "demo.lisp")
Hello world
```

Note: it "runs" the file immediately displaying the result to the console.

Car, Cdr, Cons, List

Two assembly language routines for the IBM 704 became the primitive operations for decomposing lists: car (Contents of Address Register) and cdr (Contents of Decrement Register). Lisp dialects still use car and cdr (pronounced /'k?r/ and /'k?d?r/) for the operations that return the first item in a list and the rest of the list respectively. (from wikipedia)

So how do these work?

- **car** - a function that returns the first item in the list (some Lisp dialects call it first)
- **cdr** - a function that return the rest of the list (some Lisp dialects call it last)
- **cons** - construct a new list element
- **list** – a function to create a new list from a list of elements

```
> (= x '(1 2 3 4 3 2 1))
> ( car x)
1
> (cdr x)
(2 3 4 3 2 1)
```

These of course can be nested to access any element

```
> (cdr (cdr ( cdr x)))
(4 3 2 1)
```

Also if you have an empty list ()

```
> (cdr '())
null
```

This becomes important when you create recursion functions that strip off elements as they loop.

car and **cdr** breaks lists apart - how do we stick them back together? This is where **cons** is used.

```
> (cons 'a '(1))
(a 1)
> (cons 'b x)
(b 1 2 3 4 3 2 1)
```

A more complex example is adding a list to another list

```
> (cons '(a b) '(c d))
((a b) c d)
```

So what would I get if I took car of the newly constructed list??

```
> (car '((a b) c d))
(a b)
```

A list is returned - the first element. Did you try to create like this?

```
> (cons 'a 'b)
Exception : Not a sequence
```

This is because **cons** expects its second argument to be a list (or null). So to do the above you need

```
> (cons 'a (cons 'b nil))
(a b)
```

An alternative way to this is to use the **list** function as follows

```
> (list 'a 'b)
(a b)
```

The list function builds list using cons for you – and so makes thinks a lot simpler !

Function and conditionals

When a list is processed or evaluated the first atom is treated as a function. This function can be define by the user, here's an example

```
> (def hello () (prn "hello world"))
LSharp.Function
> (hello)
hello world
"hello world"
```

So what's going on here? The first line defines a function called "hello" that takes no arguments (). The functions then calls **prn** to print with new line. Note how 'def' is a function that returns a type 'Lsharp.Function' I then invoke the function by using it in a list (hello). So that hello is evaluated. The output is "hello world", and the return value from **prn** function "hello world" is also displayed.

Here's an example of a simple function with an argument:

```
> (def addone (x) (+ x 1))
LSharp.Function
> (addone 5)
6
> (addone (addone (addone 7)))
10
```

You might try and see what happens if you **addone** to a list. To make functions useful we need one more element, the conditional. In LISP this is normally called 'cond' however in this Lisp dialect it instead uses the more familiar (to non LISP people!) if function. **if** relies on the fact that the atom 't' mean true and null is not true. So a few examples:

```
> (if null 'true 'false)
false
> (if t 'true 'false)
true
> (if (> 3 5) 'true 'false)
false
```

To use an if statement you need a predicate - a function that return true (t) or false (null), such as "is x greater than y ?" which would be written (> x y), or "is x equal to y ?" written as (is x y). b]if[/b] statement can be compound so they are very similar to the old style LISP cond statements, i.e (if a b c d e) means if a then b elseif c then d else e. A few examples:

```
> (= x 1)
1
> (if (is x 1) 'one (is x 2) 'two  'no)
one
> (= x 2)
2
> (if (is x 1) 'one (is x 2) 'two  'no)
two
> (= x 3)
3
> (if (is x 1) 'one (is x 2) 'two  'no)
no
```

So how to create a function 'spell' that uses an '**if**' statement and outputs any number up to 99 (without listing all 99 values)? We need to have multiple if statements:

```
(def spell (x)
   (progn
   (if (> x 29)
     (progn (= x (- x 30)) (pr “thirty “)))
   (if (> x 19)
     (progn (= x (- x 20)) (pr “twenty “)))
   (if  (is x 1) ‘one
        (is x 2) ‘two
        (is x 3) ‘three
        (is x 4) ‘four
        (is x 5) ‘five
        (is x 6) ‘six
        ‘no
)))

LSharp.Function
>
(spell 1)
one
> (spell 2)
two
> (spell 21)
twenty one
> (spell 24)
twenty four
> (spell 20)
twenty no
>
```

Important: look how to chain instructions you need a function - called **progn**. You can’t simply go ((pr “hello”) (pr “world”)) - can you see why ? The first element of the list isn’t a function - its a list ! So it errors. The correct way to write this is (progn (pr “hello”) (pr “world”)).

Also: “twenty no” ??? - Well it’s almost there - can you think how to improve this?

Controlling the Robot

So what is the process for controlling Robobuilder from within L#. Here's a diagram outline the basic approach.

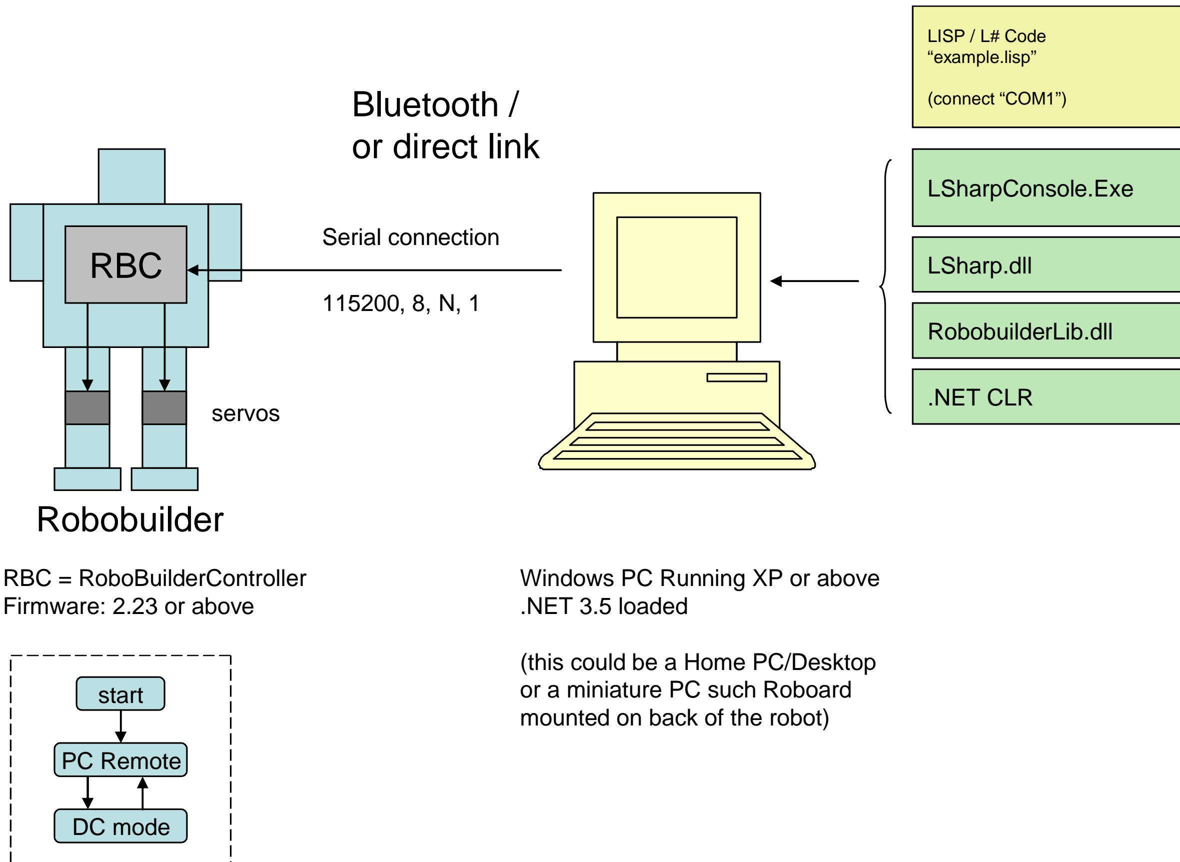


Figure 1

The LISP program or functions (in yellow) are defined in a file and loaded and run within the console application. This enables Lisp programs to access .NET dynamic libraries such as the built in .NET functions – such as System.Console, or Windows libraries – see later section, or the Robot API library – RobobuilderLib which will see more of later.

The communication with the Robot is via a serial link, this can be established using the .NET object System.IO.Port.Serial. The link can be a direct physical link or if the optional module has been added to the RBC via Bluetooth (BT). The Serial object will take care of the physical device interface and all the application needs is the name of the COM port (real or virtual) that connects to the robot.

Once connected, binary commands to the robot firmware make it possible to control either high level functions in the default PC remote mode – such as performing a built-in motion, or play a built in sound. There is also Direct Control (DC) mode that allows access directly to individual servos, so that they can be configured or controlled. Switching between modes is controlled again by a binary sequence. The RobobuilderLib provides and SDK so that the application program does not need to know the sequences to use the firmwares features.

L# provides simple access to windows functions. Here's an alternative way of creating your own **prn** function, using the built in system console class.

```
> (Console.WriteLine "helloworld")
Helloworld
```

To get the application to load the dynamic library (or assembly) Lisp has a function reference. So to access the serial port – which isn't a built in core function – we first load it and then we can

Now to create a function to connect to the robot !!

```
(reference "System.IO.Ports")
(using "System.IO.Ports")
(def connect ()
  (= sport (new "SerialPort"))  ; create new instance of serial port
  (.set_BaudRate sport 115200)  ; set its baud rate
  (.set_PortName sport "COM3")  ; specify the COM port
)
```

The function **connect** creates a new instance of a SerialPort (= sport (new "SerialPort")) and assigns ("=" function) to the variable 'sport'. In L# you can access .NET propert using .set_xxxx to access property xxxx on the object instance. The above example sets the baud rate to 115200 and port to COM3.

At this point the serial port is not connected to the robot. To do that there is a method (or function) called Open (and likewise Close to disconnect). In L# to call the method you specify its name, preceded with a '.' and provide the object instance as an argument. Here is an example session:

```
> LSharp.Function
> (connect)
null
> (.open sport)
null
> (.close sport)
null
```

Using the serial connection – we can pass it to the robobuilder lib to issue commands to the robot, for example for reading the firmware and serial number. First step is to load the library (as we did with the Serial Port), using reference. If the dll is not in the same folder as the Console you need to provide the full path RobobuilderLib.

```
(reference "RobobuilderLib")
(using "System.IO.Ports")

(def connect (pn)
  (prn "connecting to " pn)
  (= sport (new "SerialPort"))
  (.set_BaudRate sport 115200)
  (.set_PortName sport pn)
  (= pcr (new "RobobuilderLib.PCremote" sport))
)
```

This creates a serial port object (sport) and also creates a PCremote object (pcr) that takes the serial port as a parameter, to enable it to do the communication with the robot. The serial port is still not open at this point. But everything is now set up. **connect** is defined as a function that takes the serial port as an argument - such as "COM1".

To make this easier to use and read, the code can be built into a function that's lets the user select the port from the keyboard. Here it is:

```
(def askPort ()
  (with (k 0 p 0 y 0)
  (prn "Available Ports:")
  (= p (System.IO.Ports.SerialPort.GetPortNames))
  (each y p (prn y))
    (prn "Select:")
    (while (is (= k (Console.ReadLine)) "" ) (pr ": "))
     (if (not (member? k p)) (do (prn "Invalid port?") (askPort)) k)))
```

The function **askPort** will prompt the user, and get a list of possible ports using the .NET function into a local atom / variable 'p. This will contain a list i.e. "COM1" "COM2" etc. Using the **each** iterator function, its displays each element of the list returned. Finally it asks the user for input. If the input matches (is a member of the list returned) it exists with value of the data entered 'k. Otherwise its displays an error message and re-enters (a recursive call) **askPort** - so the user can try again. So a lot of code and a few elements - well it is Sunday!

We've got a connection using PCremote, so lets get data from the robot to prove its all working!. The simplest is the serial number. Using the readSN() method the function **getsn** opens the serial port - reads the serial number and then closes. sn is the return value of the function, it should really be a local variable. Note: If you are not connected at this point the system will hang - it should time out - but it's not at the moment. So make sure your serial cable is connected and the robot is on.

```
(def getsn ()
  (do
  (.open sport)
  (= sn (.readSN  pcr))
  (.close sport)
  sn)
)
```

Finally we now have the top level function that brings it all together:

```
(def run_robobuilder()
  (connect (askPort))
  (getsn)
)
```

Just type (run_robobuilder). You should see something like this:

```
> (run_robobuilder)
Available Ports:
COM1
COM3
COM9
Select:
: COM3
connecting to COM3
"1041100010***"
>
```

BTW the * are there to obscure my serial number - it's actually just 13 numbers.

More PC Remote mode functions

Download into the same directory as the LSharpConsole.exe. *Now to use this just start Lsharp*

```
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright © Rob Blackwell. All rights reserved.
> (load "Final.lisp")
................
OK
> (run_robobuilder)
Available Ports:
COM1
COM3
COM9
Select:
: COM3
connecting to COM3
Good Firmware loaded (2.26)
Serial Number = 1041100010***
Distance      = 10 cm
ok
```

So how does this work. You've seen the key functions - but I've added a few more including a top level function run_robobuilder. Lets see this:

```
(def run_robobuilder()
   (connect (askPort))
   (MessageBox.Show "make sure robot is connected to serial port and on"  "warning" )
   (.open sport)
   (checkver)
   (prn "Serial Number = " (getsn))
   (prn "Distance      = " (readdistance) " cm")
   (.close sport)
   'ok
)
```

It starts by setting up the connection to a serial port as we have already seen. It then pops up a message box to warn you to connect and switch on the robot. It then opens the serial connections (.open sport) and then calls a function to check the firmware version you are running.

```
(def checkver ()
   (if (not (.Isopen sport))
   (.open sport))
   (= v (.readVer pcr) )
   (if (< v 2.23)
     (prn "Download new firmware")
     (prn "Good Firmware loaded (" v ")" )
    )
    V
)
```

This routine get the firmware via PCremote using this (= v (.readVer pcr)) storing the resultant string in a variable v. Note although v is a string it can still be used for a numeric test as L# converts seamlessly between string and number.

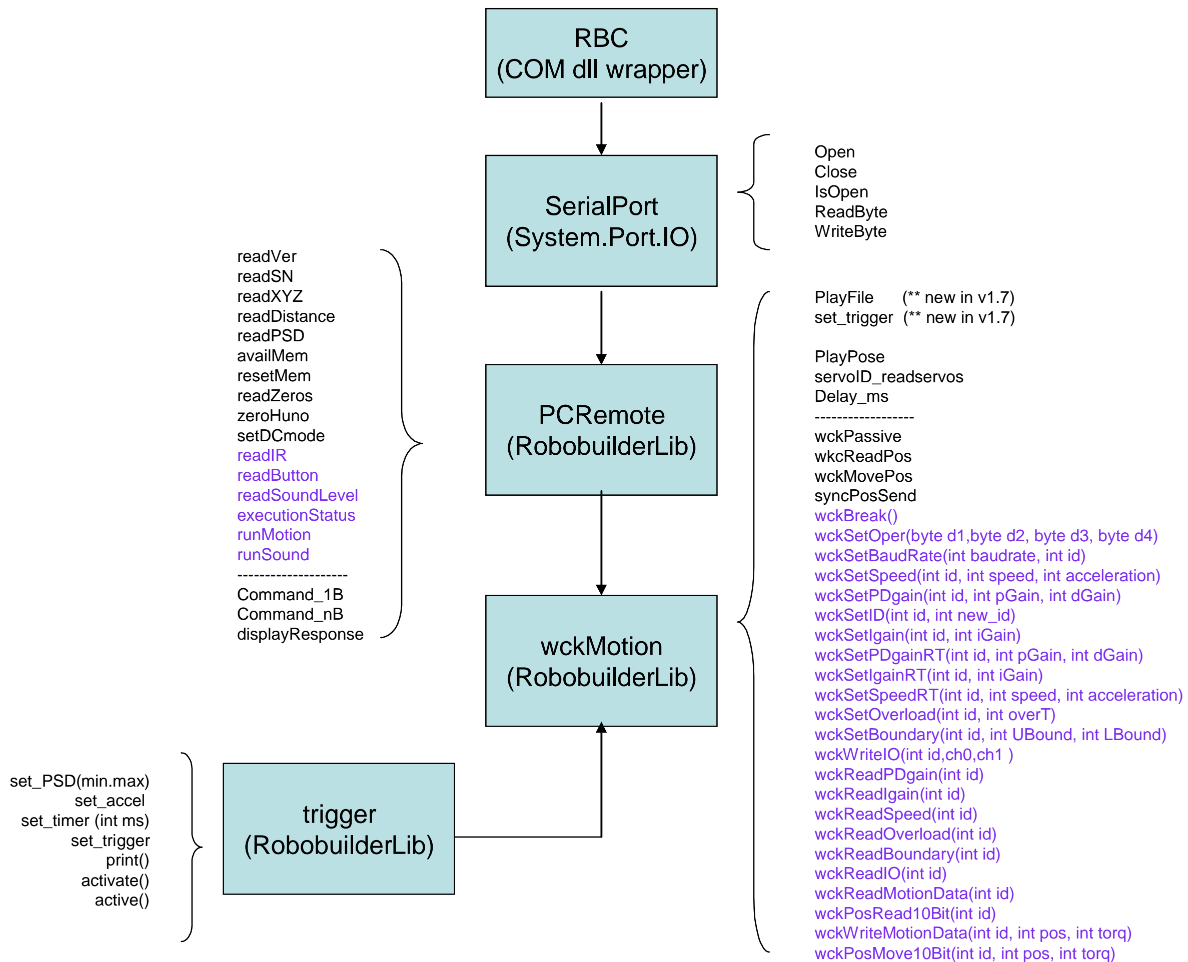


Figure 2- PC Remote and wckMotion schema

Some of the PC remote functions do rely on recent firmware I suggest 2.23 or better go to the http://Robobuilder.net/eng site to download the latest.

The program then gets and displays (using the print (pr) command) the serial number and also now the distance. If you don't have a distance sensor it shows 10 (cm).

Final.lisp also includes **runMotion** and **runSound**. More on these tomorrow - but for now assuming your robot is still on and connected

```
> (runMotion 7)
```

This will make robot take up basic pose.

Menu driven motions

If you have it the code running and connected to the robot you can now try using the built in motions, As mentioned yesterday to play a motion use the function (runMotion n) where n is a number. The numbers are defined in the RBC_over_serial_protocol_1.13.pdf - see *http://robosavvy.com/forum/viewtopic.php?t=3503*

I've added the motions specified into a list along with a description.

```
(= menu '( 1 "GET UP A"
           2 "GET UP B"
           3 "TURN LEFT"
           4 "MOVE FORWARD"
           5 "TURN RIGHT"
           6 "MOVE LEFT"
           7 "BASIC POSTURE"
           8 "MOVE RIGHT"
           9 "ATTACK LEFT"
          10 "MOVE BACKWARDS"
          11 "ATTACK RIGHT"
           0 "EXIT"))
```

As you can see I've called the list 'menu' which should give a good clue what's coming next. By modifying the code that is used to select the COM port I have a function that will display the menu and request input.

```
(def ask (prompt error menu)
  (with (k 0 )
  (prn prompt)
  (each x (pair menu) (prn (car x) " - " (cdr x)))
  (while (is (= k (Console.ReadLine)) "" ) (pr ": "))
  (= k (coerce k "Int32"))
  (if (not (member? k menu))
    (do (prn error) (ask menu))
   )
  k
  )
)
```

How does this work? The menu is a list of items '(1 a 2 b) etc.. The function [b]pair[/b] converts this into another list, of pairs of items, i.e. (pair '(1 a 2 b)) result is '((1 a) (2 b)). This list is then used by the **each** function so that x is first set to '(1 a) and then '(2 b) etc. The print function **prn** then prints the first element of the value of x using **car**. Since (car '(1 a)) is '1. and then " - " and then the rest of the list using **cdr**, i.e. (cdr '(1 a)) would be 'a. It then goes on to treat each pair in the same way. This results in the menu being printed. The function then requests input and looks if that matches an member of 'menu list. If it does it converts the text input to a number (using the **coerce** function) which becomes the return value of the function. Notice if I look in the list for "1" it doesn't match with a '1 because one is string and the other Int32.

A simple while loop looking for the user to press zero to exit or any number to call the motion can then be created as follows

```
;; loop until 0 selected
(def domenu ()
(with (item 1)
(while (not (is item 0))
(= item (ask "Choice :" "No such option" menu))
```

```
(if (> item 0) (runMotion item))
      )
  )
)
```

See how its calls the (ask) function passing in the prompt string and menu as a list - this could be used as generic function for any picklist. The value returned by **ask** is stored in ‘item and then passed to **runMotion**. This could equally be used to **playSound** in the same way.

Assuming you have loaded Day7.lisp and (run_robobuilder), now enter (domenu) and you can select and play any of the built-in motions on robobuilder. i.e.

```
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright © Rob Blackwell. All rights reserved.
> (load "Day7.lisp")
> (run_robobuilder)
> (domenu)
```

Isn’t Lisp/L# lovely and compact!

Accessing the accelerometer.

One of the optional sensors that can be purchased with Robobuilder is a 3-axis accelerometer (Robosavvys' Xmas edition bundles it along with the distance sensor). The sensor is wired into the RBC control box. The values of the accelerometer can be accessed using the PCremote library via a method readXYZ. The values can then be stored into a symbol xyz using (= xyz (.readXYZ pcr)) You must first have loaded *Day7.lisp* and (run_robobuilder). The symbol 'xyz will contain a list of 3 numbers i.e. '(4 3 -6) representing the X, Y and Z values. To access X you use **car** i.e. (car xyz) which gives 4 in this example. By chaining the **car** with **cdr** you can access the other elements i.e. y is (car (cdr xyz)) and z would be (car (cdr (cdr xyz))). In LSharp an alternative way is to use the **nth** function which lets you access elements of a list directly. So to get x would be (nth xyz 0) and y would be (nth xyz 1). Notice the first element is index zero.

```
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright © Rob Blackwell. All rights reserved.
> (load "Final.lisp")
................
OK
> (run_robobuilder)
Available Ports:
COM1
COM3
Select:
: COM3
connecting to COM3
Latest Firmware loaded (2.26)
Serial Number = 1041100010***
Distance      = 10 cm
ok
> (.open sport)
> (= xyz (.readXYZ pcr))
System.Int32[]
> (= xyz (tolist (.readXYZ pcr)))
(6 -15 65)
> (= xyz (.readXYZ pcr))
System.Int32[]
> (nth xyz 1)
17
```

Notice how the data returned by PCremote is actually an array of int32s. They can be converted into a list or use directly. **nth** works equally well on either. It also works on strings - so (nth "hello" 3) is character "l".

Of course this has only read one value - to sample the data its needs a read in a loop with a delay, using another command sleep where (sleep x) to pause the program for x seconds. Here is a function **readSensors** that loops for 10 times and every 0.5 secs reads both XYZ values and the distance sensor and outputs the result.

```
(def readSensors ()
   (if (not (.isopen sport)) (.open sport)) ;; ensure com port open
   (do (prn "i   X   Y   Z   Distance")
   (for i 0 10
     (do (= xyz (.readXYZ pcr))
       (prn i " " (nth xyz 0) "," (nth xyz 1) "," (nth xyz 2) " : " (.readdistance pcr))
       (sleep 0.5))
     )
   )
)
```

```
> (readSensors)
i  X  Y  Z  Distance
0 2,-17,61 : 10
1 1,-17,64 : 10
2 -1,-20,60 : 10
3 3,-14,64 : 10
4 -3,-1,63 : 10
5 -1,25,53 : 10
6 0,32,56 : 10
7 1,30,50 : 10
8 -7,10,62 : 10
9 -5,-15,63 : 10
10 -1,-15,65 : 10
null
```

In this demo - the distance (PSD) sensor is not connected - hence the function always returns 10 back. Using this, the data can be stored, or an application could check the data and if the XYZ values falls outside a certain range invoke a motion to balance the robobuilder dynamically. Of course stock motions won't be sufficient we need direct control of the servos ....

Access the servo directly

To read and control the wck servos requires switch into Direct Control mode on Robobuilder. A PC Remote binary sequence switches into this mode and then subsequent binary command are sent directly onto the wck servo bus. This means every aspect of a servo can be controlled including setting boundary conditions, PID values, even the servos ID and baudrate. The individual positions of servos can be read or can be moved to specified positions. The servos can be put into continuous rotation mode or switched to passive input mode. Using these commands each individual servo can be addressed and controlled using a unique ID between 0-31, or with the synchronous move they can all be commanded at once. To make this easy within the windows environment is the wcKMotion class. This class can be accessed like any other .NET class and is part of the RobobuilderLib.dll. The class takes as a parameter of its constructor the PCremote class used in previous examples. In the same way PCremote required SerialPort to be passed to it, wcKMotion requires an instance of PCremote. This is the code to do the set up.

```
(def dcmodeOn ()
  (if (not (.isopen sport)) (.open sport))
  (= wck (new "RobobuilderLib.wckMotion" pcr)))
)
```

The code first checks the serial port isOpen - and if not opens it. This is needed as the wckMotion issues the control codes to switch on DC mode when created. When DCmode is on you cannot issue PCremote commands - as they will be passed to the servos and treated as gibberish. The PCremote/wckMotion classes should protect you from this in that it will auto switch - and turn DCmode off. You can tell when the RBC is in DC mode as the amber light will come on on the RBC unit - the box at the back of the robot. Once in DCmode it's simple to access a servos positions using wckReadPos method.

```
(def getServoPos (n)
   (.wckReadPos wck n)
   (cadr (.respnse wck)) ;; could use nth here!
)
```

The wckReadPos method returns a Boolean - true if successful and puts the returned data into a two element array called **respnse**. The first element is the load (generally this zero) and second element is the position. Hence the code used **car** and **cdr** to return the position of second element. So to read the position of servo id 5, all is now required is the command (getServoPos 5). The servo can also be put into passive mode - so that it can be moved into different positions:

```
(def setPassive (n)
   (.wckPassive wck n)
)
```

This can now be combined with a **getServoPos** to read the servos as they are moved - the start of creating a capture and play capability. Here's an example session where I set servo 12 (left arm at elbow) to passive and then read 20 times - every ½ second - the values off the servo as I move it.

```
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright © Rob Blackwell. All rights reserved.
> (load "Day7.lisp")
................
OK
> (run_robobuilder)
Available Ports:
COM1
COM3
Select:
: COM3
connecting to COM3
Latest Firmware loaded (2.26)
Serial Number = 1041100010***
Distance      = 10 cm
ok
> (def dcmodeOn ()
    (if (not (.isopen sport)) (.open sport))
    (= wck (new "RobobuilderLib.wckMotion" pcr))
   )
LSharp.Function
> (def dcmodeOff ()
    (.Close wck)
)
LSharp.Function
> (def setPassive (n)
    (.wckPassive wck n))
LSharp.Function
> (def getServoPos (n)
    (.wckReadPos wck n)
    (cadr (.respnse wck)))
LSharp.Function
> (dcmodeOn)
RobobuilderLib.wckMotion
> (setPassive 12)
True
> (for i 0 20 (do (prn (getServoPos 12)) (sleep 0.5)))
100
99
100
99
45
114
... etc
```

How to play servo moves

The method to move a servo to a specified position is wckMovePos. This takes three parameters, the id of the servo the position of the servo on the torq setting. These values are explained in the wckProgramming guide. Here's a Lisp function that the calls the method assuming that wck has been created using **dcmodeOn** as explained yesterday.

```
(def setServoPos (id pos torq)
  (.wckMovePos wck id pos torq)
  (cadr (.respnse wck))
)
```

So to use this simply enter (setServoPos 12 120 2) and the servo will move. We could some extra tests to ensure the parameters are with in bounds. Also notice the return value is in the repnse data. This reports the position as the command is received by the servo. Repeated reads will then update the actual position as the servo attempts to move to its requested position.

Using the code yesterday its straightforward to create a function to read and capture the data from the **getServoPos** function as follows:

```
> (setServoPos 12 60 2)(capture 12 5 0.05)
20
20
28
43
57
60
60
```

```
(= data ())

(def capture (id n t)
    "capture n points every t secs"
  (for i 1 n
    (do (= data (cons (prn (getServoPos id)) data) )
      (sleep t)
     )
  )
  "Captured")
```

Notice how capture stores the servo position in a list called data. Each new element is added using cons function however this then means the last element is at the front and the first at the back of the list. Now we have a list of positions we can play them back to the servo using the **setServoPos** function just created. First we need to use reverse function to convert the list into the order needed and then we play it back.

```
(def play (id n t)
  (for i 0 (- n 1)
   (do (pr ".")
       (setServoPos id (nth data i) 2)
       (sleep t))
  )
"Complete"
)
```

We could slow or speed up the motion using a different t parameter, or even play it back via a different servo - so get the left arm to mirror the right arm for instance. Here’s an example session:

```
> (setPassive 12)
True
> (capture 12 10 0.5)
45
45
45
73
61
46
29
38
62
62
"Captured"
> (= data (reverse data))
(45 45 45 73 61 46 29 38 62 62)
> (play 12 10 0.5)
.........."Complete"
```

This is all works well for one servo (and could be extended) but to move all the servo at the same time - for a complex motion requires a separate function.

Synchronous moves

The method to send an array of servo positions to move synchronously is called .**SyncPosSend**. A lisp function provides an easy to use wrapper around wckMotion class. The code assumes were using the global wck which will need to be initialised by **dcmodeOn** before this function can be called.

```
(def setSyncMove (lastid torq position)
(.SyncPosSend wck lastid torq position 0))
```

So to use this code it needs three parameters - the last id in the list - so for16 servo (0-15) last id would be 15. The torq is the speed it moves; a value between 0-2; 0 is the fastest. Finally it expects an array of at least last id +1 values in an array. So to create an array of positions for basic pose for instance would look like this.

```
(= basic '(143 179 198 83 106 106 69 48 167 141 47 47 49 199 204 204))
(setSyncMove 15 2 (toarray basic))
```

Caution: this will immediately move to basic pose, very rapidly.

Notice the **toarray** function, this is required to convert a list into an array object[] so it can be passed into the wck method.

Now what is really needed is to move there slowly in gradual steps, from where we are now, to where we want to be, in this case the basic pose. To achieve this we must first create an array of our current position

```
(def getallServos(n)
(with (current () )
(for i 0 n
   (= current (cons (getServoPos (- n i)) current))
 )))
```

This function is a simple iterative loop reading each element into Assuming Day7.lisp is loaded and (run_robobuilder) and (dcmodeOn). See how it uses cons to construct the output list.

```
> (getServoPos 2)
248
> (getServoPos 1)
184
> (getServoPos 0)
146
> (cons (getServoPos 0) (cons (getServoPos 1) (cons (getServoPos 2) ())))
(146 184 248)

;;or all 15 in one go !
(= cur (getallServos 15))
(146 184 248 46 109 102 60 4 205 142 23 47 62 228 204 204)
```

So we have two lists, current position cur - and our final potion basic. We need to calculate the in between points. So for example if servo is at position 20 and it moves to 60 in 10 steps, then each step is (60-20)/10 or 4. Id we can then move the servo in the following way 20 24 28 32 36 40 ... we will have a smooth move. You could off course use non-linear algorithm here such as a sinusoidal that make the movement greatest in the middle part of the move.

Smooth moves and the In-betweens

To generate a smooth move between to sets of servo positions it is necessary to calculate the difference or distance between the two lists. Lets take a simpler example of just three servos. The current position is represented by the list ‘(6 3 2) and the final position of the servos the list ‘(3 1 1). To evenly step between the two lists requires each servo to move by a different amount - an interval. To move in 5 equal steps would require a list formed from (6-3)/5, (3-1)/5, (2-1)/5. Heres the code to create that list:

```
(def md (xs ys step)
  "calc distance between two list"
  (= list ())
  (for i 0 (- (len xs) 1)
    (= t  (/ (- (nth xs i)  (nth ys i)) step))
    (= list (cons t list))
  )
  (reverse list)
)
```

So the command (md ‘(6 3 2) ‘(3 1 1) 5.0) will result in an output (0.6 0.4 0.2). Notice the step must be a floating point (or decimal) number. If it’s a whole or integer number it will get the wrong result. 6/3 = 0 in integers, but 6/3 =0.5 in decimal. Lisp will convert integers to decimals if part of the calculation is decimal, but if all parts are integers the result will be integer. See how list is built by successive **cons** commands as in **getAllServos** from yesterday.

We can use the **interval** list created to calculate the in between moves by adding it successively to the start list. Here the complete function:

```
(def smove (a b n)
  "smooth move position a to b in n steps"
  (= interval (md a b (* n 1.0))) ;; calculate distance (= l (len a))
  (for i 0 n
    (= list ())
    (for j 0 (- l 1)
      (= t  (- (nth a j)  (* (nth interval j) i)) )
      (= list (cons t list))
    )
    (prn "(setSyncMove " l " " 2 " '(" (reverse list) "))")
   ;(setSyncMove l 2 (reverse list)) (sleep 0.1)
  )
)
```

Here is an example of it running..

```
> (smove '(6 3 2) '(3 1 1) 5)
(setSyncMove 3 2 '(6   3   2)  )
(setSyncMove 3 2 '(5.4 2.6 1.8))
(setSyncMove 3 2 '(4.8 2.2 1.6))
(setSyncMove 3 2 '(4.2 1.8 1.4))
(setSyncMove 3 2 '(3.6 1.4 1.2))
(setSyncMove 3 2 '(3   1   1)  )
```

If the output looks correct then un-comment the **SetSyncMove** line (just remove the ;) and you’re ready for smooth move! To get the bot to smoothly take up basic position, apply the command (smove cur basic 10). This assumes you have followed yesterdays post and set up **cur** to be a list of the current servo positions and **basic** is the predefined basic pose list. If this is working you can now create your own motions and poses. The **sleep** will control the speed of the move. It set at 0.1s so a 10 step interval will take 1s to complete. This could also be passed as a parameter to the function.

Making complete motions

Yesterdays post showed how to move between to points A and B using a function to divide into equal step the movement of each servo and then play that out with a specific delay. To create complex motions all is that is now required is to create a list of points that make the motion up, with addition of how long and how many steps to take. If we look at the example of "Punch Left" this can be easily encoded as follows:

```
(= initpos ‘( 125 179 199  88 108 126  72  49 163 141  51  47  49 199 205 205 ))
(= Data0   ‘( 125 179 199  88 108 126  72  49 163 141  51  47  49 199 205 205 ))
(= Scene1  ‘( 107 164 233 106  95 108  80  29 155 129  56  62  40 166 206 208 ))
(= Scene2  ‘( 107 164 233 106  95 145  74  40 163 154 117 124 114 166 206 208 ))
(= Scene4  ‘( 126 164 222 100 107 125  80  29 155 142  79  44  40 166 206 208 ))
(= punchleft (list initpos
(list  1  70 Data0)
(list  8 310 Scene1)
(list  1 420 Scene2)
(list  5 200 Scene4)
(list 15 300 Data0)))
```

The individual points are specified as lists - in Robobuilder speak each is a "scene". A motion such as punchleft is created by stringing together a number of scenes, in this example called Data0, Scene1 etc. These can be created using MotionBuilder software that comes with the bot. The extra two number 1 70 for example specify the number of in between step (or frames in Robobuilder speak) and the total time take. So for 1 frame in 70ms that 70ms, if it were 2 frames that would be 35ms per frame.

The following function **playmotion** will take the list ‘punchleft and play it in real-time as if it had been upload. It uses a recursive subroutine called play1 which works by playing the top potion on its list and then calling itself with the remaining list until the playlist is empty.

```
(def playmotion (c x)
  "Play a motion list"
  (= ip (car x))
  ;; move from current pos to ip
  (prn "Do initial move")
  (prl ip)
  ; move to initial position smoothly ..
  (smove c ip 10 0.05)
  ;;now loop
  (play1 ip (cdr x))
)
(def play1 (c x)
  "used by play move c -> x"
  (if (or x)
  ( do ;move through list
     (prn "Do next move")
     (= cur (car x))
     (= nof (car cur))
     (= trt (/ (cadr cur) nof))
     (= gt  (car (cdr (cdr cur))))
     (smove c gt nof (/ trt 1000.0) )        ;smooth move from c to gt
     ;recurse
     (play1 gt (cdr x))
    )
   ;else we've finished
   (prn "Done")
  ))
```

So to use this function we must first load and run final.lisp and **run_robobuilder** to connect the serial port. This also needs the other functions covered in the last chapters.

```
(load "wckutils18.lisp")
(demo)
```

This is great, but having the servo data stores as code is not easy to manage, what would make things easier is if they can be loaded from a file.

Loading data from CSV files

If we want to manage a lot of motion data - the best way to access would be if the positions were stored in a file. Robobuilder use a file form called rbm, but here I've used Comma separated values (CSV). The big plus is that you can then manipulate the data easily in Excel or Openoffice calc. There are many ways to read a CSV files using .NET libraries, the method used here uses a simple character based parsing looking for commas and then assumes the cell contains a number. So first we need to create a CSV file using a favourite text editor (or spreadsheet such OpenOffice Calc). Here is an example using the "punch left" data from previous section.

Example test.csv

```
#comments
0,1,125,179,199,88,108,126,72,49,163,141,51,47,49,199,205,205
70,1,125,179,199,88,108,126,72,49,163,141,51,47,49,199,205,205
310,8,107,164,233,106,95,108,80,29,155,129,56,62,40,166,206,208
420,1,107,164,233,106,95,145,74,40,163,154,117,124,114,166,206,208
200,5,126,164,222,100,107,125,80,29,155,142,79,44,40,166,206,208
300,15,125,179,199,88,108,126,72,49,163,141,51,47,49,199,205,205
```

The format of the CSV file is very straightforward, any line that starts with a '#' is trated as a comment and ignored. The first column is the duration the motion should take and the second column is the number of intermediate steps. So 10 step in 250 ms represents each step taking 25ms. Each row is a motion that starts from where the last one finished. The first row uses the current position as its starting point (by doing a read servo position). The remain columns represent the servo position for each corresponding servo starting with servo ID 0 in column 2 (or C).

To read this data requires two functions. The first **readcsv** takes a filename as a parameter and then uses the .NET function File.ReadAllLines to create a string array, one line per record. So f will be a list containing '("line1" "line2" "line3") for example. Using for the **each** command then sets line to each row intern. If the line starts with a # the line is treated as a comment and ignored, in the example above the #comment line is ignored. The line is passed to the second function **splitline** that separates the entries using the comma into a list of numeric fields. **splitline** also separates out the first two entries so that it follows the format used yesterday. It also defaults any empty cells to zero.

```
(def readcsv (filename)
 (with (f "" z ())
  (= f (System.IO.File.ReadAllLines filename))
  (each line f
   (if (not (is #\# (car line)) )
     (do (= z (cons (splitline line) z)))
    )
   )
   (reverse z)
 )
)

def splitline (text)
(with (l "0" z () r 0)
   (each a (toarray text )
      (= a (str a))
      (if (is a  ",")
          (do (= z (cons (coerce l "Int32") z)) (= l "0") )
          ;else
          (not (is a " ")) (= l (+ l a))
```

```
        )
    )
    (= z (cons (coerce l "Int32") z)) ;; last arg
    (= r (reverse z))
    (cons (cadr r) (cons (car r) (list (cdr (cdr r)))))
))
```

Here is an example of it in action.

```
> (= data (readcsv "test.csv"))
((1 0 (125 179 199 88 108 126 72 49 163 141 51 47 49 199 205 205)) (1 70 (125 179 199 88 108 126 72 49 163 141 51 47 49
199 205 205)) (310 8 (107 164 233 106 95 108 80 29 155 129 56 62 40 166 206 208)) (1 420 (107 164 233 106 95 145 74 40 1
63 154 117 124 114 166 206 208)) (5 200 (126 164 222 100 107 125 80 29 155 142 79 44 40 166 206 208)) (15 300  (125 179 1
99 88 108 126 72 49 163 141 51 47 49 199 205 205)))
```

The first entry needs the two zero stripped out and this is achieved using a combination of : *(car (cdr (cdr (car data))))*. The result can then be passed to the **playmotion** as outlined yesterday by constructing a new list called punchleft

```
(= punchleft (cons (car (cdr ( cdr (car data)))) (cdr data)))
(playmotion cur punchleft)
```

So using L# it is possible to create pre-programmed motions - stored in CSV file format and then loaded on demand and played in real time. L# is not only capable of text and list processing but through use of .NET functions can manipulate graphics and windows to enable real time display of the robot status.

To simplify the work need to play a CSV file, there is (since V1.7 of RobobuilderLib) a built-in function PlayFile that will take read a file sending each line to the Robot and calculating (or interpolating) the in-between positions. In addition there is a trigger mechanism that enable the application to set up pre-configured limits which when reach cause the motion to stop.

In the following example firstly the file test.csv is played (assuming wck has been previously created as an instance of a wckMotion.). Then it creates a trigger that is set to stop the motion if the value of the PSD sensor is outside of the range 30-60 cm. The PSD sensor is read every 260 ms. The trigger is set against the instance of wckMotion object but only affects the motion if activated.

Here’s example code fragment. The first line plays the file without a trigger set. Then the code creates a trigger object for the distance sensor to fire when outside the range 30-60cm. It the set the trigger against the wck object (already assumed to be created) and then activates the trigger and replays the file.

```
(.Playfile wck "test.csv")

  (= trg (new "RobobuilderLib.trigger"))
  (.set_PSD   trg 30 60)
  (.set_timer trg 250)
  (.print     trg)
  (.set_trigger wck trg)
  (.activate  trg true)

 (.Playfile wck "test.csv")
```

Other triggers will be able to be set, but currently it only supports PSD sensor and the accelerometer, although the object has been designed to cover the IR sensor and Sound sensor.

Controlling the IO on a servo

Robobuilder servos come in to two forms, black and transparent with colour LEDs. Using the RobobuilderLib library you can directly control the LEDs. If you have the black form – you can also use the as readable input ports – need to read the wck manual for details. To support writing to LEDs the library provides the method wckWriteIO. To use from within L# the code is as follows:

```
(load "final.lisp")                ; load PC remote routines
(load "wckutils18.lisp")               ; load wck mode routines

(def wckwriteIO (n c0 c1)
        "turn IO on/off"
        (if (not (bound 'wck)) (dcmodeOn))
        (.wckWriteIO wck i c0 c1)
     )

(def setallLeds (n c0 c1)       ;
        "turn on/off all servo leds"
        (for i 0 n
             (wckwriteIO i c0 c1)
          )
)

(run_robobuilder)               ; initialise connection to robot
(dcmodeOn)                      ; switch into DC mode

(wckWriteIO 10 true true)       ; set both LED on servo 10 on
(setallLeds 16 true false)      ; set all C1 high = red
(sleep 1)                       ; wait 1s
(setallLeds 16 false true)      ; set all C2 high = blue
(dcmodeOff)                     ; exit DC mode
```

Windows based graphics using L# !

To create a graphics window using .NET is straightforward in L#. The function **createwindow** creates an instance of a Windows Form object. The global variable form1 is the handle for the window. This creates the frame and standard windows controls such as min, max and close. In this example they won't work as they won't have functions behind them. Within the form the code creates a Panel object for the graphics to be drawing on. The exact size can be passed into the application. It could also add buttons, labels, menus etc using the same technique. Once the panel is created it then gets a handle to the graphics object which is used to send graphic primitive commands too.

```
(reference "System.Windows.Forms"
           "System.Drawing")
    (using    "System.Drawing"
           "System.Windows.Forms")

(def createwindow (title x y)
     (do
      (= form1 (new "Form"))
      (.set_text form1 title)
      (.set_autosize form1 true)

      ( = pb (new "System.Windows.Forms.Panel"))
      (.set_size    pb (new "Size" x y))
      (.set_Location pb (new "Point" 24 16))
      (.add (.controls form1) pb)

      (= g (.CreateGraphics pb))
     )
)
```

Notice how **using** and **reference** take multiple arguments. Once this has been created its possible to draw into the object using the Graphic handle (in this case **g**). All the .NET draw methods are available such as lines, arc, curves, polygons, rectangles etc. The following function **demo** draws x and y axes and then a circle at the coordinates specified. It also displays text passed to it. You can see it sets up different colour pens, red and black, for different draw commands. It also sets up a Font for the text. The background is set white by the clear method.

```
(def demo( txt x y )
     (= w (.width pb))
     (= h (.height pb))
     (= h2 (/ h 2))
     (= w2 (/ w 2))
     (= x (+ x w2))
     (= y (- h2 y))
     (.clear g (Color.FromName "White"))
     (= axis (new "Pen" (Color.FromName "Black")))
     (= pen  (new "Pen" (Color.FromName "Red")))
     (= font (new "Font" "Arial" (coerce 8.25 "Single" )))
     (.drawline    g axis 0 h2 w h2)
     (.drawline    g axis w2 0 w2 h)
     (.drawellipse g pen (- x 6)  (- y 6) 14 14)
     (.drawstring  g txt font (.Brush pen) (new "PointF" 10 10))
    )
```

The **demo** function can auto detect the size of the window by reading the width and height properties. Now to run the demo. All the code necessary is here in this post - no extra files are

required today. First there must be a window visible using (.show) before **demo** is called. The window doesn't save or cache any information so it must be at the front.

```
(createwindow "Demo" 250 250)  ; create graphics window 250x250 pixels
(.show form1)                  ; display window
(demo "test" 0 0)              ; draw demo - cross with circle and text
; admire output :)
(.close form1)                 ; finished
```

This will display an image that looks like this –

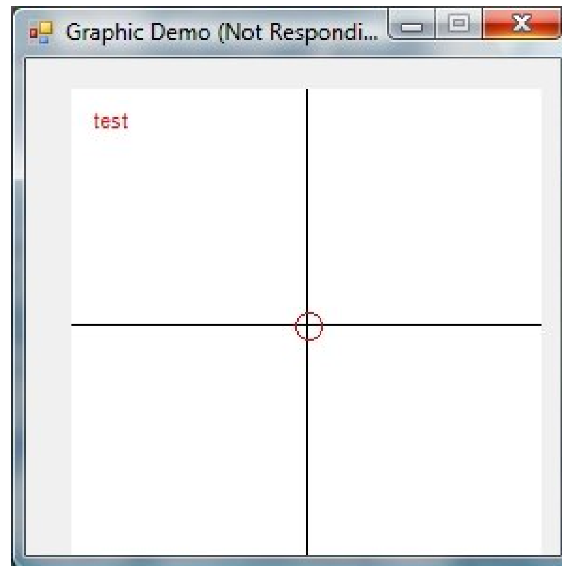


Figure 1

Look to see what happens if you add multiple demo calls with different x and y values. This will give you a clue as to tomorrows post.

Plotting and animation

Building on the windows display of yesterday, it's simple to generate an animated real-time display of data. To demonstrate this is a function called **anim** which displays an oscilloscope style plot of a sine wave. L# itself doesn't have maths routines, but it can access the C# directly. To calculate the Sin of an angle (in radians) it can call the Math class.

```
L Sharp 2.0.0.0 on 2.0.50727.3603
Copyright (c) Rob Blackwell. All rights reserved.
> (= Pi 3.1415)
3.1415
> (Math.Sin (/ Pi 2))
0.999999998926914
> (* 100 (Math.Sin (/ Pi 2)))
99.9999998926914
```

It could access PI directly as well using Math.Pi.

The demo relies on two main functions, **plot** and **drawlist**. The **plot** function is essentially the same as **demo** from yesterdays post. **drawlist** takes a list of x-y coordinates and creates a line linking the points together. An example would be (drawlist g '((0 0) (20 20) (30 120) (140 20) (50 50)) "Red") which draws a red line of 5 segments. The **g** is the handle to the graphics object created by **createwindow** explained yesterday. **drawlist** recursively calls **drawline** to draw the lines. As each call is made is calls its self with one less element on the list until the list is empty, which is tested for by the logic function **and**. Here is the code:

```
(def drawlist (g l c)
    (= from (car  l))
    (= to   (cadr l))
    (if (and from to)
      (do
      (drawline g (car from) (cadr from) (car to) (cadr to) c)
      (drawlist g (cdr l) c)
      )
    )
)
(def drawline (g fx fy tx ty c)
    (= lp (new "Pen" (Color.FromName c)))
    (= w (.width pb))
    (= h (.height pb))
    (= h2 (/ h 2))
    (= w2 (/ w 2))
    (= fx (+ (coerce fx "Int32") w2))
    (= fy (- h2 (coerce fy "Int32")))
    (= tx (+ (coerce tx "Int32") w2))
    (= ty (- h2 (coerce ty "Int32")))
    (.drawline  g lp fx fy tx ty) )
```

The main function is **anim**. It loops or iterates through values of 'n *until the 'q' key is pressed*. This is achieved by **exit?**. It checks the Console.keyavailable property. As the code loops it calculates the position of a a sin wave and passes that to the **plot** via **scope** routine. This also creates the text label with the co-ordinates. In the main loop there is also a log of all the points that it has generated in a variable called **history**. It takes the last three items from **history** and puts them into a variable **h** using : ((car history) (cadr history) (car (cdr (cdr history)))). This is passed to **drawline** to create the blue tail.

```
(def exit? ()
  (if (Console.keyavailable)
    (if (is (.key (Console.ReadKey true)) (ConsoleKey.Q)) (err "Quit Pressed"))))

(def scope(x y)
  (= x (coerce x "Int32"))
  (= y (coerce y "Int32"))
  (= text (+ "(" (str x) " " (str y) ")" ))
  (plot text x y)
)

(def anim (f t s)
 (createwindow "Demo" 250 250)
 (.show form1)
 (while true
 (do
 (= history ())
 (= n -125)
 (while (< n 125)
    (= a (* 100 (Math.Sin (/ (* n Pi) f))))
    (scope n a)

    (= history (cons (list n a) history))
    (= h (list (car history) (cadr history) (car (cdr (cdr history)))))
    (drawlist g h "Blue")

    (sleep t)
    (= n (+ s n))
    (exit?)
 )
 ))
 (.hide form1)
)
```

To run enter the frequency, the time and the resolution and watch it go! The smaller the time value the faster the dot moves across the screen. Here's a few examples.

```
> (anim 50 .5 5)
Exception : Quit Pressed
> (anim 100 .1 10)
Exception : Quit Pressed
>  (.show form1)(drawlist g history "Blue")
```

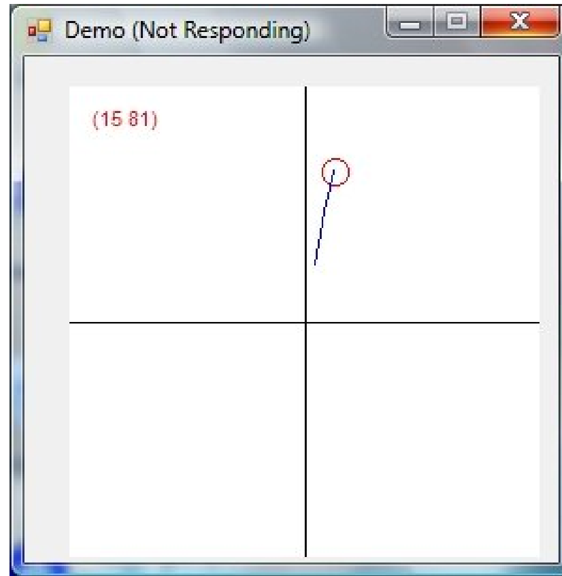


Figure2

To stop it press q at the console window. You can look at the variable history which has the points plotted on the current pass - this can also be display by passing it to **drawlist** (see last line in above example).

Real-time accelerometer

Plotting data in real-time from your Robobuilder robot. It's important to know if we have a connection to the robot. So the following code does a couple of tests **serial?** and **remote?** to see a connection has been made. They look to see if the global variables exist; **sport** - which is the handle for the serial port, and **pcr** the handle to the RobobuilderLib.PCremote class. The functions return the name of a colour which is then passed to a display function called **status**.

```
(def serial? () (if (and (bound 'sport) (.isopen sport)) "Green" "Red"))
(def remote? () (if (and (bound 'pcr) (is "Green" (serial?))) "Green" "Red"))

(def status(x y txt c)
 (= pen  (new "Pen" (Color.FromName c)))
 (.fillellipse g (.Brush pen) (- x 10)  (- y 6) 14 14)
 (= pen  (new "Pen" (Color.FromName "Black")))
 (.drawellipse g pen (- x 10)  (- y 6) 14 14)
 (.drawstring  g txt font (.Brush pen) (new "PointF" (+ x 10) (- y 4)))
)

(def demo2(f g)
 (createwindow "status" 250 250)
 (.show f)
 (.clear g (Color.FromName "White") )
 (status 205 8  "Serial"  (serial?) )
 (status 160 8  "Remote"  (remote?) )
)
```

If you type (demo2) a window will appear with a couple of red indicators showing not connected yet. You will need to use the functions created from the last few days post. The function **status** creates the filled circles using the **fillelipse** rather than the **drawelipse** method. Also the **bound?** function enables to test to see if a variable has been created or set. It returns false if it doesn't exist. Very useful!

To debug code, it's simpler not to have the robot connected, so I simulate the accelerometer using random data. The function **readAcc** will return the '(X Y Z) values as a list and so by replacing this with a function that generates random numbers means the plot routine can be tested easily. The random numbers are generated using the C# class Random which when instantiated returns random number using the .Next method. (.next r 50) generates a random integer between 0-50. The function then subtracts 25 to generate random number between -25 and +25. It does this 3 times to create a list i.e. '(10 -5 20).

```
(= r (new "Random"))
(def readAcc ()
   (list (- (.next r 50) 25)  (- (.next r 50) 25) (- (.next r 50) 25) ))
;replace with this for actual sensor reading
;(def readAcc () (.readXYZ pcr))
```

Now for the main code for **plotaccel**. It's very similar to the **anim** routine. It displays the values but also displays the serial status - showing if the robot is connected - although the serial port can't seem to detect if you turn the robot off!

```
(def plotaccel (r)
  "Plot - sample rate rHz"
  (= r (/ 1.0 r))

  (with (acc 0)
   (createwindow "Accelerometer Demo" 250 250)
   (.show form1)
   (= n 1)
   (while (< n 1000)
     (= acc (readAcc))
     (scope (car acc) (cadr acc))
     (status 205 8  "Serial" (serial?))
     (status 160 8  "Remote" (remote?))
     (drawlist g '((-125  40) (125   40)) "Black") ; limit
     (drawlist g '((-125 -40) (125 -40))  "Black") ; limit

     (= history (cons (list (car acc) (cadr acc)) history))
     (= h (list (car history) (cadr history) (car (cdr (cdr history)))))
     (drawlist g h "Blue")

     (sleep r)
     (= n (+ 1 n))
     (exit?)
   )
   (.hide form1)
  )
)
```

To run simply enter (plotaccel 5) the value 5 represents the frequency the accelerometer is read and plotted. 5Hz works well. If the routine works with dummy **readAcc** routine then load Day7.lisp and (run_robobuilder) and connect to the COM3 port. Replace the **readAcc** with a function to read the accelerometer as described in earlier post (day 9). The **scope** function draws the axes and plots the point. It is passed in this example X and Y values. But you could pass any 2 of the three values. The **plotaccel** function also displays a couple of boundary lines - these represent the tipping points (or could do) of the robot. The idea would be when the accelerometer values cross the line would be the point to activate a balance routine for instance.

When running, move your robobuilder bot around and watch the circle move - its great fun!

Property lists

Property list are a powerful feature of Lisp. All through this series atoms have essential one value. It might be a string or a number or list but it is in essence a single thing. The idea behind property list is to give atoms properties that can be stored and read. So if an atom can be any vehicle, a car or bike for instance, it could also have other properties such as colour. Back in the 60's this was ground breaking, but with the rise of object oriented languages, today this concept is common place and of course core to .NET, so much so that L# doesn't implement property lists. But of course with Lisp it’s easy to define our own functions to do the same and at the same time use powerful .NET feature to build it.

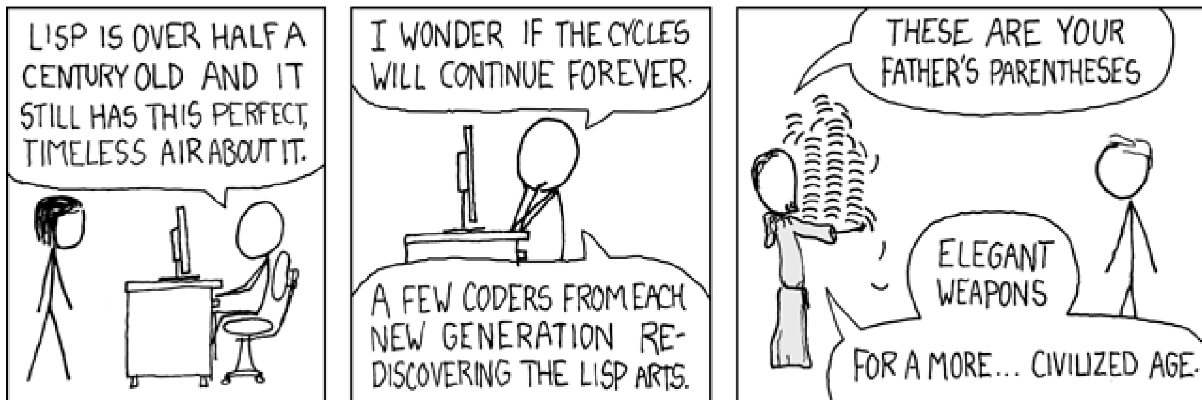
Here are the key functions:

```
(def props ()
 (if (no (bound 'propdb)) (= propdb (new "System.Collections.Hashtable")) propdb)
)

(def putprop (i v p)
  (props)
  (do
   (if (no (.contains propdb i))
       (.add propdb i (new "System.Collections.Hashtable")) )
   (if (.contains (.Item propdb i) p) (.set_Item (.item propdb i) p v)
     (.add (.Item propdb i) p v))
   v)
)

(def get (i p)
   (props)
   (.item (.item propdb i) p)
)

(def remprop (i p)
  (props)
  (putprop i null p)
)

(def prprop (i)
  (props)
  (with (x (.item propdb i))
    (each y (.keys x) (Console.Writeline "{0,14} = {1}" y  (.item x y)))
  )
)
```

A brief explanation of the functions. Property lists are global - they have no local scope so the function **props** ensures the global list is created first time it’s used. **putprop** assigns the property value to the atom. The way its implemented it here is as a hashtable. So each atom has its own hashtable which contain the properties and their values. **get** retrieves the property value and **remprop** will remove it (actually it just assigns it to null). So what can we do with it? The final function is **prprop** which prints out all the properties and their values for a particular atom. Going back to our example of vehicles we can create two vehicles C1 and C2 - each with a set of unique properties which can then be retrieved when required.

```
(putprop 'C1  'RED   'COLOUR)
(putprop 'C1  'CAR   'TYPE)
(putprop 'C1  'FORD  'MAKE)

(putprop 'C2  'BLUE  'COLOUR)
(putprop 'C2  'BIKE  'TYPE)
(putprop 'C2  'BMW   'MAKE)

> (get 'C1 'COLOUR)
RED
> (get 'C2 'COLOUR)
BLUE
> (prprop 'C2)
     MAKE = BMW
   COLOUR = BLUE
     TYPE = CAR
> (remprop 'C2 'TYPE)
null
> (prprop 'C2)
     MAKE   = BMW
   COLOUR = BLUE
     TYPE   =
```

Blockworld / SHRDLU



(from xkcd)

Property lists make storing and manipulating information easy(ier). One of the most famous AI programs written is SHRDLU by Terry Winograd in 1972 and it was written in LISP – see http://hci.stanford.edu/~winograd/shrdlu/index.html for more info and even the source code for the program. It allows for a dialog between a user and the computer about a virtual world of blocks resting on either each other or table. Here is a simple graphic of Blockworld.

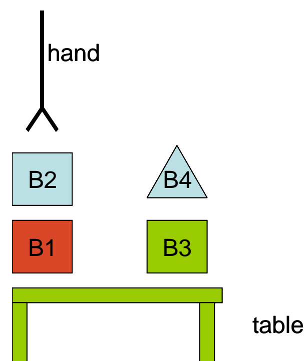


Figure 3

What SHRDLU attempts to do is allow a user to ask the question how do I put block B1 on B3. This requires realising that B2 rests on B1 and B4 rests on B3 etc. SHRDLU encapsulates this data in property lists as follows:

```
(putprop 'B1 'BLOCK   'TYPE)
(putprop 'B1 'TABLE   'SUPPORTED-BY)
(putprop 'B1 '(B2)    'DIRECTLY-SUPPORTS)
(putprop 'B1 'RED     'COLOUR)

(putprop 'B2 'BLOCK   'TYPE)
(putprop 'B2 '(1 1 2) 'POSITION)
(putprop 'B2 'B1      'SUPPORTED-BY)
(putprop 'B2 'BLUE    'COLOUR)
```

Notice how B1 is supported by the table and supports B2, whilst B2 is supported by B1. It then uses functions to manipulate that data.

The idea behind 'blockworld' was to enable the user to have a dialog with the computer. To enable this, the program specifies a world consisting of different shape on a table, and a set of functions that can manipulate those shapes based on a set of rules. So for example

```
((def ungrasp (obj)
    (remprop 'HAND 'GRASPING)
    (= plan (cons (list 'ungrasp obj) plan )))

(def grasp (obj)
    (putprop 'HAND obj 'GRASPING)
    (= plan (cons (list 'grasp obj) plan )))

(def movehand (pos)
     (putprop 'HAND pos 'POSITION)
     (= plan (cons (list 'movehand pos) plan )))

(def putat (obj place)
     (do   (grasp obj)
           (moveobj obj place)
           (ungrasp obj)))

(def moveobj (obj place)
      (movehand place))
```

grasp will set the property of the HAND so that its grasping the object specified. Similarly with **ungrasp** it clears the property of the hand and **movehand** changes the position of the hand. The code also tracks what actions have been taken the atom **plan**.

Here is an example session:

```
> (grasp 'B1)                  ;; get hold of the Block
((grasp B1))
> (get 'HAND 'GRASPING)  ;; check if the property has been updated
B1
> (ungrasp 'B1)                ;; now let go of block
((ungrasp B1) (grasp B1))
> (get 'HAND 'GRASPING)  ;; check the property again
null

;; reset the plan
(= plan ())

> (putat 'B1 '(1 1 1))         ;; combined action
((ungrasp B1) (movehand (1 1 1)) (grasp B1))

> (reverse plan)             ;; plan needs to be reversed
((grasp B1) (movehand (1 1 1)) (ungrasp B1))
>
```

If we ask the system to move an object (putat 'B1 '(1 1 1)) it creates a plan of actions grasp, moveobj, ungrasp. But what if the space being moved to was taken? And how do we update the virtual world model? This requires a more sophisticated model which can take into account when

one object B1 is put onto B2 then it also changes the supports. So for instance B1 is supported by the TABLE and B2 supported by B1. If we move B2 we need to remove the B2 SUPPORTED-BY property on B1 as it isn't any longer, but also remove the DIRECTLY-SUPPORTS property on B1. Both objects need updating. In this model one object can DIRECTLY-SUPPORTS multiple objects - so this requires the **del** function which removes matching elements from a list:

```
(def del (e l)
  (if (not l) null
     (is e (car l)) (del e (cdr l)) (cons (car l) (del e (cdr l)))))

; see how it removes all 'HEADS from list leaving just TAILS
> (del 'HEAD '(HEAD TAIL HEAD TAIL TAIL))
(TAIL TAIL TAIL)

(def removesupport (obj)
  (putprop (= sup (get obj 'SUPPORTED-BY)) (del obj (get sup 'DIRECTLY-SUPPORTS))  'DIRECTLY-
SUPPORTS)
  (putprop obj null 'SUPPORTED-BY)
)

; --------------------------------------------------------
;  BEFORE
;
> (prprop 'B1)
      DIRECTLY-SUPPORTS = B2
      TYPE = BLOCK
      COLOUR = RED
      SUPPORTED-BY = TABLE

> (prprop 'B2)
     POSITION = (1 1 2)
     TYPE = BLOCK
     COLOUR = BLUE
     SUPPORTED-BY = B1

> (removesupport 'B2)

;  --------------------------------------------------------
;  AFTER
;
> (prprop 'B1)
    DIRECTLY-SUPPORTS =
    TYPE = BLOCK
    COLOUR = RED
    SUPPORTED-BY = TABLE

> (prprop 'B2)
    POSITION = (1 1 2)
    TYPE = BLOCK
    COLOUR = BLUE
    SUPPORTED-BY =
```

So after the (removesupport 'B2) both B1 and B2 have been updated. If we had tried to move B1 the system would detect that its DIRECTLY-SUPPORTS 'B2. And so it would then first attempt to move B2 before changing the support on B1. Similarly you can create **addsupport**. Its a bit more complex. It needs to check if the object can be supported. So in Blockworld whilst a BLOCK can support another BLOCK, an object of type BALL or PYRAMID can't. So you couldn't put B1 on B4 (the pyramid - see yesterdays diagram)

By adding more functions (I'll make my simple version "asdfg.lisp" available in the final

download) you can create a set of functions that allow a command (puton 'B1 'B3) which then creates a plan of how to put one block on top of another. In the full SHRDLU you can have a dialog along the lines:

```
> PICK UP THE RED BLOCK
: WHICH RED BLOCK - THERE IS MORE THAN 1
> PICK UP B2
: OK
> PUT ON B3
: (GRASP B4)
: (MOVE TO TABLE)
: (UNGRASP B4)
: (GRASP B2 BLOCK)
: (MOVE HAND TO B3 BLOCK)
: (UNGRASP B2)
: OK
>
```

Of course this requires natural language parser which s beyond the scope of these posts (look at the link to SHDRLU in yesterdays post for more details and the original source code!).

Why is this useful? Because for robots to do more the play pre-programmed moves they need the ability to plan ahead. This went out of fashion during the 90's as top down AI was replaced with bottom up AI based on sensory input, but may be its time for a comeback?

Searching and matching

Doctor, Doctor I think I'm suffering from Deja Vu!
Didn't I see you yesterday?

A useful feature is to compare two lists, or to search one list for matching entries in a second list. The modern way to do this would be to use regular expression searching but its easy to add a function **match** to do this in LISP (LSharp.NET) with lists and it also shows how this type of matching works.

The following **match** function takes two lists and if they identical returns true, but if they're different returns null. It also will treat ">" as a wild card that will match with any *ONE* character. This makes the matching far more powerful. Here's the code with some examples:

```
(def match ( p d)
  (if (and (empty? p) (empty? d)) t
      (or  (empty? p) (empty? d)) nil
      (or  (is (car p) '>)  (is (car p) (car d)))
        (match (cdr p) (cdr d))
))

;;examples

(match '( a b c)  '(a b c))
True
> (match '( a b c)  '(a b d))
null
> (match '( a > c)  '(a b c))
True
```

Doctor, Doctor I keep thinking there is two of me
One at a time please

So a simple match is fine, but what if we want to match multiple items. This requires a recursive search. When the atom "+" is identified it then continues to match the remaining items. Here is the code:

```
(def match ( p d)
  (if (and (empty? p) (empty? d)) t
      (or  (empty? p) (empty? d)) nil
      (or  (is (car p) '>)  (is (car p) (car d)))
        (match (cdr p) (cdr d))
      (is  (car p) '+)
        (if (match (cdr p) (cdr d)) true (match p (cdr d)))
   )
)
;; more examples

> (match '( a > c)  '(a b d c))
null
> (match '( a + c)  '(a b d c))
True
>
```

So in this example the "+" matched with the b and d, and the function can then return true as the whole list is then matched. This is useful, but it would be even better if it not only did it find the match but also what items in the list matched - and this leads to the final version of **match**

```
;utilities
(def atomcar (z) (str (car (str z))))
(def atomcdr (z) (str (cdr (str z))))
(def setstr  (x y)  (LSharp.Runtime.VarSet (LSharp.Symbol.FromName x) y environment))
(def evalstr (x)    (eval (LSharp.Symbol.FromName x)))

;match function
(def match ( p d)
  (if (and (empty? p) (empty? d)) t
      (or  (empty? p) (empty? d)) nil
      (or  (is (car p) '>)  (is (car p) (car d)))
         (match (cdr p) (cdr d))
      (and (is (atomcar (car p)) ">") (match (cdr p) (cdr d)))
         (setstr (atomcdr (car p)) (car d))
      (is  (car p) '+)
        (if (match (cdr p) (cdr d)) true (match p (cdr d)))
      (is (atomcar (car p)) "+")
         (if (match (cdr p) (cdr d)) (setstr (atomcdr (car p)) (list (car d)))
             (match p (cdr d))         (setstr (atomcdr (car p)) (cons (car d) (evalstr (atomcdr (car p))))
         )
      )
  )
)
;;even more examples

> (match '( a >L c) '(a b c))
b
> (match '( a +G c) '(a b d c))
(b d)
```

This has introduced a couple of extra functions, **atomcar** which returns the first character of an atom. So (atomcar 'HELLO) returns "H", whilst (atomcdr 'HELLO) return "ELLO". **setstr** is the string equivalent of (= H 0) i.e. (setstr "H" 0) will assign the atom H the value 0. The new **match** will now allow a symbol to be appended to the search character ">L", which if the match succeeds is assigned the values that are matched. In the above examples the L is assigned b and the G assigned (b d). Very powerful!

Doctor, Doctor I feel like a spoon!
Well sit still and don't stir!

So now we have matching we recreate one of the most well known AI programs of all time, Eliza the psychiatrist. http://en.wikipedia.org/wiki/ELIZA.

From Wikipedia: *First implemented in Weizenbaum's own SLIP list-processing language, ELIZA worked by simple parsing and substitution of key words into canned phrases. Depending upon the initial entries by the user the illusion of a human writer could be instantly dispelled, or could continue through several interchanges. It was sometimes so convincing that there are many anecdotes about people becoming very emotionally caught up in dealing with DOCTOR for several minutes until the machine's true lack of understanding became apparent*

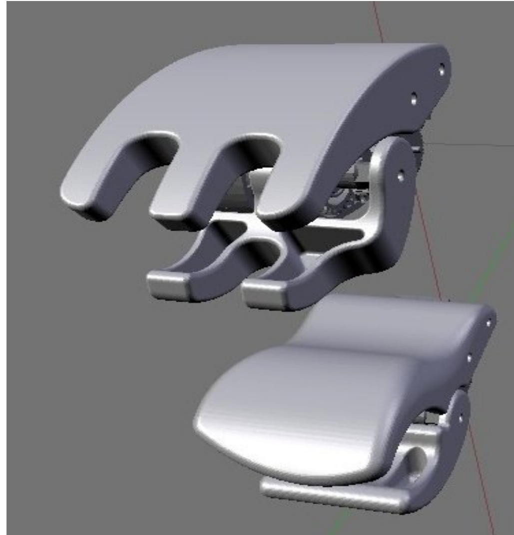
```
(def evalstring (s)  (eval (LSharp.Runtime.ReadFromString s)))

(def doctor ()
  "Demo"
  (with (item 1)
    (while (not (is item 0))
        (do
          (while (is (= inp (Console.ReadLine)) "" ) (pr ": ")))
          (= inpt (evalstring (+ "" (.ToUpper inp))))
         (if
           (iso inpt 'BYE)
              (do (prn "GOODBYE" )(err "Done"))

           (match '(I AM WORRIED ABOUT +L) inpt)
              (prn "WHY ARE YOU WORRIED ABOUT " L "?")

           (match '(+ MOTHER +) inpt)
              (prn "TELL ME MORE ABOUT YOUR FAMILY")

           (prn "TELL ME MORE")
          )
         (pr ": ") ) ) )
; sample session

> (doctor)
: (HELLO)
TELL ME MORE
: (I AM WORRIED ABOUT FRED)
WHY ARE YOU WORRIED ABOUT FRED?
: (ITS HIS MOTHER REALLY)
TELL ME MORE ABOUT YOUR FAMILY
: BYE
GOODBYE
```

The code uses the match function to search pre-canned phrases and uses the assignment mechanism to the work some of the question back into the response - which is where the program gets is ability mimic understanding. The user input is converted into a list using **evalstring**. So this takes an input string such as "(this is input)" into '(THIS IS INPUT). To make this more powerful and believable requires far more match rules. If you go to this link http://norvig.com/paip/eliza.lisp you will see a much more developed version written in 1991 by Peter Novig in LISP.

Doctor, Doctor Can I have second opinion?
Of course, come back tomorrow!

Adding a gripper with dynamic control of a servo

The Robobuilder humanoid robot can be extended with up to 32 servos (from the standard basic kit of 16). One way of extending is to add some form of gripper to enable the robot to pick up objects in its world. A simple gripper can be built in many ways; one way is to use a 3D printed object such as is available from Shapeways.Com. Here is an example of a couple available:



Top – New style gripper, Bottom original style

```
(def opengripper (d)
  (with (cnt 0 delta 0 cp (coerce (getServoPos 18) "Int32") np 0 delta 5)
    (if (< cp widepos) (= cp widepos))
    (while (and (< cnt maxit) (> (Math.abs (- cp widepos)) 2) (> delta 0))
    (do
      (= cnt (+ cnt 1))
      (prn cnt ", " cp ", " delta)
      (setServoPos 18 (- cp d) 4)
      (sleep dt)
      (= np (coerce (getServoPos 18) "Int32"))
      (= delta (Math.abs (- cp np)))
      (= cp np)
    ))
))
(def closegripper (d)
  (with (cnt 0 delta 0 cp (coerce (getServoPos 18) "Int32") np 0 delta 5 )
    (if (> cp closepos) (= cp closepos))
    (while (and (< cnt maxit) (> (Math.abs (- cp closepos)) 2) (> delta 2))
    (do
      (= cnt (+ cnt 1))
      (prn cnt ", " cp ", " delta)
      (setServoPos 18 (+ d cp) 4)
      (sleep dt)
      (= np (coerce (getServoPos 18) "Int32"))
      (= delta (Math.abs (- cp np)))
      (= cp np)
    ))
 ))
```

In the above examples the gripper is connected to servo ID 18. The gripper is widest open at "widepos" and closed at the smallest value "closepos". The basic process is, that rather than simply moving the servo between those two position, the code steadily moves to the desired position in increments (delta) and after each increment checks to see what its actual position is. If after several attempts it doesn't make the target position it assumes the hand has closed on an object and stops.

```
;Example
(setServoPos 18 70 4)      ; set initial position of gripper
(closegripper 5)
(opengripper 5)
```

A'mazing

In the Robobuilder C programming tutorial there is an application written in C to enable the robot to traverse a maze using the distance sensor. You have to compile it in AVR C and download to the RBC. To make this easier to do, I've converted it to L#/Lsharp.Net.

The code builds a map on screen of the maze as the robot moves around. It uses the distance sensor to identify the walls and then the following algorithm:

1. If there is no wall, move forward.
2. If the wall is detected, then check left side.
3. If front and left side wall are detected, then check right side.
4. If front, left and right side walls are detected, go back the opposite way.

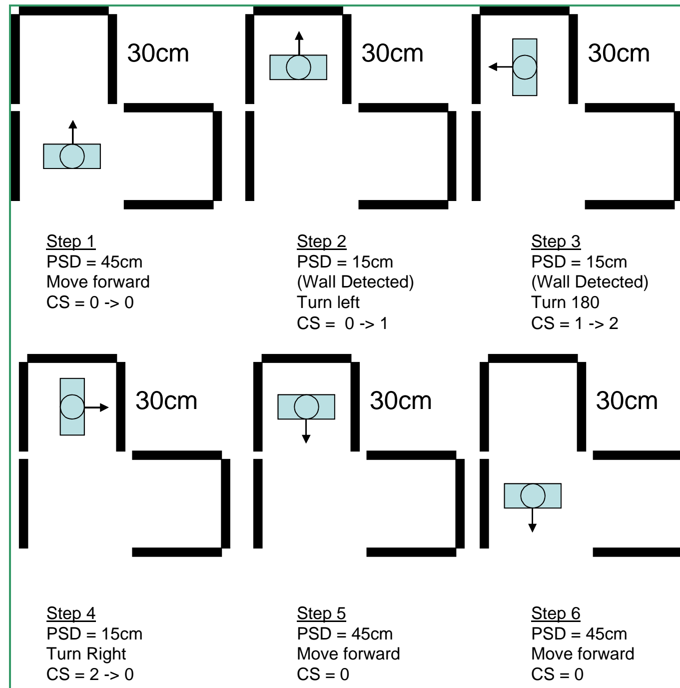


Figure 4

You can run it in a simulate mode without the robot. Here's an example of it in action:

```
> (maze)
  ........
  ........
  ........
  ........
  ....^...
  ........
  ........
  ........
  distance ?
  : 20
  LT90
  Run motion 3
  Run motion 3
```

```
Run motion 3
........
........
........
........
....<...
........
........
........
```

The program defines a number of simple primitive motions based on the built in motions:

```
(def leftturn90   ()     (prn "LT90")     ( runMotion 3) (runMotion 3) (runMotion 3))
(def leftturn180 ()      (prn "LT180")   ( runMotion 3) (runMotion 3) (runMotion 3) (runMotion 3) (runMotion 3))
(def rightturn90 ()      (prn "RT90")     ( runMotion 5) (runMotion 5) (runMotion 5) (runMotion 5))
(def forward      ()     (prn "FWD")      ( runMotion 4) (runMotion 4) (runMotion 5) )
(def back         ()      (prn "BACK")  ( runMotion 10))
```

The map is created using a simple string to represent 8x8 grid. The functions **putmap** and **showmap** display and show the robot as a simple turtle type character in the maze. Each square in the maze represent a 30x30cm cell. A '.' indicates if the cell is empty and a '*' if the cell has been visited.

The function **maze** then uses these routines to solve the maze, by walking the robot around, and measuring the distances to the nearest wall using the PSD.

The code is written so that it uses two key routines from final.lisp **runMotion** that plays the motion turn left etc and **readdistance** that reads the PSD sensor. These routines are initially defined in maze2.lisp to allow user to simulate by entering the distance and then printing the motion to be called - so it can be run without a robot attached.

To run fully all that is required is to load final.lisp which will redefine **runMotion** and **readdistance** to be the actual routines. Call (run_robobuilder) and connect the PC to robot. Select "Basic Posture" and then exit the menu. Now type (maze) and off it goes, showing its its progress on the map!

```
(def maze ()
      (= px 4) (= py 4)          ; assume center of maze
      (= pd 0)                  ; facing up
      (= pt '("^" ">" "v" "<"))  ; maze turtle character
      (= cs 0)                  ; current state = 0
      (clrmap)                   ; reset map

      (while true
        (= pd (mod pd 4))
        (putmap px py (nth pt pd)) (showmap)
        (= d (readDistance))
        (if  (> 12 d)              (back)
            (and (is cs 0) (> 30 d))   (do (leftturn90)  (= pd (+ pd 3)) (= cs 1))
            (and (is cs 1) (> 30 d))   (do (leftturn180) (= pd (+ pd 2)) (= cs 2))
            (and (is cs 2) (> 30 d))   (do (rightturn90) (= pd (+ pd 1)) (= cs 0))

            (do
              (= cs 0)
              (forward)

              ; update map
              (putmap px py "*")
```

```
            (= pd (mod pd 4))
            (if (is pd 0) (= py (- py 1))
                (is pd 1) (= px (+ px 1))
                (is pd 2) (= py (+ py 1))
                (is pd 3) (= px (- px 1)))
            (if (< px 0) (= px 0) (< py 0) (= py 0) (> px 7) (= px 7) (> py 7) (= px 7))
           )
        )
      )
    )
```

You may well have to tune the motion routines to optimise performance. You could add a test to see if the robot has fallen over - and if it does, get it to stand up, reset the maze and start again.

Joystick

It's possible to use DirectX libraries and L# to read a joystick. The joystick is an input device that can have many different features, for instance analogue or digital style, varying numbers of buttons and even double or triple axis, vibration, tilt. All these parameters are controllable via the device handle. So the first piece of code gets the user to select the device to be used. **askjoy** is similar to the serialPort selected in that its lists the devices returned and then ask the user to input a number corresponding to device required. Here's the code with an example of it running. You need **ask** function from Day7.lisp.

```
(reference "Microsoft.DirectX.DirectInput")
(using "Microsoft.DirectX.DirectInput")

(def askjoy ()
 "Pick joystick from List"
 (with ( n 1 menu '(0 Exit))
  (= gl (Manager.GetDevices (DeviceClass.GameControl) (EnumDevicesFlags.AttachedOnly) ))

  (each l gl (
      do
      (= menu (cons n (cons  (.InstanceName l) menu)))
      (= n (+ n 1)))
  )
  (= n (ask "Select Joystick Device" "No such option" menu))

  ;This picks the item from the list
  (if (is n 0) null (nth gl (- n 1)))
 )
)

> (askjoy)
Select Joystick Device
1 - ?
0 - Exit
: 1
Usage: 5
UsagePage: 1
FFDriverGuid: 00000000-0000-0000-0000-000000000000
ProductName: ?
InstanceName: ?
DeviceSubType: 2
DeviceType: Gamepad
ProductGuid: 990207b5-0000-0000-0000-504944564944
InstanceGuid: bcb9fe70-eb3a-11de-8001-444553540000
```

The joystick I have is a very simple 2 axis, 8 button digital XY input (i.e. off or on!) controller. Its device name is "?" - Not very useful! Once the handle is selected the device (like a serialport) needs to be acquired so that this application (and no other) has exclusive use. The **setjoy** function does this and displays the capabilities of the device. In the example you can see it outputs that its a 2 axis, 8 button controller. It also sets a global handle **jd**.

```
(def setjoy (l)
 "Acquire joystick for use by application"
 ;This creates the Device handle
 (= jd (new "Device" (.InstanceGuid l)))

 ;This specifies its a Joystick
```

```
  (.SetDataFormat jd (DeviceDataFormat.Joystick))

  ;This grabs it for your application
  (.Acquire jd)

  ;This lets you see the capabilities
  ( = cps (.Caps jd))
  (prn "Acquired - axes=" (.NumberAxes cps) " buttons=" (.NumberButtons cps))
)

> (setjoy j)
Acquired - axes=2 buttons=8
8
```

The joystick is now ready to be used. To read the button state the code must first call .poll method. The current state can then be read which is an object containing all the current values of the controller. The button states are available via the .getbuttons method. This returns a list, each element corresponding to a different button, first being button1 etc. If the button is pressed its value is 128, else its 0 (I assume some buttons can now be analogue as well and so this value may also vary). The function **readjoy** looks to see if a button or axis has been set. The **getButton** function waits until a button is pressed and then waits until its let go (this is known as de-bouncing). It then returns the value of the button pressed using a list **rconmap** to translate to a meaningful value:

```
(= rconmap '((0 Button1) (1 Button2) (2 Button3) ))
(= rconx   '((0 Left) (65535 Right)))
(= rcony   '((0 Up)   (65535 Down )))
(def readjoy(t l)
  (.poll jd)
  (= cs (.CurrentJoystickstate jd))

  (= but (tolist (.getbuttons (.CurrentJoystickstate jd))))

  (= r null)
  (if (is t "BUTTON")
      (each b l
        (if (> (nth but (car b)) 0) (= r (cadr b)))
      )
     (is t "X")
      (each b l
        (if (is (.X cs) (car b) )   (= r (cadr b)))
      )
     (is t "Y")
      (each b l
        (if (is (.Y cs) (car b) )   (= r (cadr b)))
      )
  )
  r
)
(def getButton ()
 (with (r () a ())
   (while (not a) (= a (readjoy "BUTTON" rconmap)))
   (= r a)
   (while (or a) (= a (readjoy "BUTTON" rconmap)))
   r
)))

> (getButton)
Button2
>
```

In the example above when **getButton** is called it will wait until Button2 is pressed (and let go) and then displayed "Button2". Similar code can be written to handle the XY axis or all 3 simultaneously. There also a lookup table (but not the code) to do the translation. With the axis button the value varies from 0 min (left or up), 32767 (center) and 65535 max - (right or down). Again these can be translated into meaningful strings. The code can then use an **if** statement to wait for a button press and then play the appropriate motion in the same way the code work in Day7 except it uses keyboard input.

Speech synthesis and recognition!

If you are running this with .NET V3 (standard with Vista and above) then you have access to the built in Speech libraries.

```
(reference "C:\\Program Files\\Reference Assemblies\\Microsoft\\Framework\\v3.0\\System.Speech.dll")

(using "System.Speech.Synthesis")

(= sp (new "SpeechSynthesizer"))

;(.SelectVoice sp "Microsoft Anna")

;; Display a greeting depending on the hour
(if (< (.Hour (DateTime.Now)) 12) (= greet "Good Morning")
        (< (.Hour (DateTime.Now)) 18) (= greet  "Good Afternoon")
        (= greet "Good Evening"))


(.Speak sp greet)
```

This code with speak a greeting depending on the time of day. It firsts references the .NET library System.Speech.dll. It then creates an instance of s speech synthesiser. If you have multiple voice available you can set those up using selectvoice method. In the above example it then determines the current time (using DateTime.Now) and generates the appropriate string. This sting is then spoken using the .Speak method.

The speech library also provides a library for speech recognition. The following code sets up the recognition engine to listen for 4 words – stand, open, close, exit. The words are specified in a string and then split into a string array object which is passed to a grammarbuilder. This .NET class simplifies building simple grammar. For more complex examples you can pass in an XML file specifying the grammar.

```
(reference "C:\\Program Files\\Reference Assemblies\\Microsoft\\Framework\\v3.0\\System.Speech.dll")

(using "System.Speech.Recognition")

(.currentUIculture (Threading.Thread.CurrentThread))

(.set_currentUIculture (Threading.Thread.CurrentThread)
              (new "System.Globalization.CultureInfo" "en-GB"))

(= menu "stand:open:close:exit")

(= c (.split menu (.tochararray ":")))
(= g (new "Grammar" (new "GrammarBuilder" (new "Choices" c))))

(= rec (new "SpeechRecognitionEngine"))

(.loadgrammar rec g)

(.SetInputToDefaultAudioDevice rec)

(.Text (.Recognize rec))
```

The recogniser can have an option timeout specified using a TimeSpan object

Useful features and functions

Variable number of arguments

So far all user created function have had a fixed number of arguments, such as (addone 5) but some of the inbuilt functions like + take a variable number of arguments (+ 1 2 3). User functions can do the same using "&" This will wrap all the following arguments up in a single list. If there are mandatory arguments they must precede the & i.e. (a b & c) means there must be at least two arguments a and b, and all remaining args are assigned to c. here a couple of examples:

```
> (def test (& x)   (map prn x))
> (test '(a b c d))
abcd
((a b c d))
> (test 'a 'b 'c 'd)
a
b
c
d
(a b c d)
> (test)
null
```

In the final version of **run_robobuilder** optional arguments can be passed so that the COM port is preselected and menus are by-passed.

macros

These are similar to variable argument commands, but with a major difference. The arguments aren't evaluated - instead the macro is expanded and then the functions are evaluated. This enables macros to generate code that a function couldn't. Here is an example of a macro used to create a repeat function:

```
(mac repeat (n & body) `(for x 1 ,n ,@body))
LSharp.Macro
> (repeat 5 (prn "hello world"))
hello world
hello world
hello world
hello world
hello world
```

Exceptions

err is a base function that throws an exception. Exceptions are built into .NET code, but err is actually a very old LISP command - and was defined back in LISP 1.6 days. The principle is the same - an exception causes the code to stop what its doing and return to where it was called immediately. Ideally you would be able to "catch" the error but V2 of Lsharp/L# doesn't seem to have implemented this (or errset in old style LISP).

pause / getch

Its often useful to read a character from the terminal. if an entire line is required - up to the CRLF then use Console.Readline, but for single character there is Console.ReadKey. Depending on what is required, there is ReadKey which waits for a key press (and optionally echoes it). The value of

the key pressed can then be read using either "keychar" or "key". "keychar" gives the ASCII values whilst "key" gives the ConsoleKey value.

The pause function uses getch to loop until a 'p' is pressed - echoing a '.' for any other key typed. It uses the "keychar" method to read the ASCII value. There is also KeyAvailable method which is set true when a key has been pressed. This enables a non-blocking read - in the example of the **exit?** function it uses this to see if a user want to break out of a loop - so if a key is available its read - and if its a Q, an exception is thrown using **err** command explained above. Notice the use of ConsoleKey.Q so it doesn't matter if it upper or lowercase (or even ctrl Q!)

```
(def getch () (.keychar (Console.ReadKey true)))

(def pause ()
  (prn "Paused - press 'p' to continue")
  ( while (not (is (getch) #\p)) (pr "."))
)

(def exit? ()
   (if (Console.keyavailable)
     (if (is (.key (Console.ReadKey true)) (ConsoleKey.Q)) (err "Quit Pressed"))))
```

time

time - times any function given to it. Very useful for seeing the performance of the link between the PC and robobuilder for instance.

```
> (time (prn "hello"))
hello
time: 2 msec
"hello"
> (time (prn "hello"))
hello
time: 1 msec
"hello"
```

map

map is a function that applies its first argument (which must be a function) to each element in the list supplied as its second argument. (In L# this seems to only work with the first list supplied - in LISP it should apply to every list supplied. A couple of examples, firstly using a predefined function such as **prn** and second using a user defined function **addone**.

```
> (map prn '(a b c))
a
b
c
(a b c)

(def addone (x) (+ x 1))
> (addone 6)
7
> (map addone '(2 4 8))
(3 5 9)
```

I use map in Blockworld code in a previous post where I want to display the properties of all the elements used: (map prprop '(B1 B2 B3 B4 TABLE HAND))

Vectors

One might think of vector maths as being for a different type of programming language but given vectors are just lists Lisp is actually very good at them. To do the vector product of two vectors (a b c) (d e f) gives the result (a*d+b*e+c*f). This can be implemented as a simple recursive function. And if we have the dot product we can normalise a vector by taking the square root of the dot product of a vector with itself. Vector routines enable the output of the accelerometer to be processed easily.

```
(def dot-product (a b)
  "calculate dot product of two vectors."
  (if (or (empty? a) (empty? b))
     0
     (+ (* (first a) (first b))
     (dot-product (rest a) (rest b)))
  )
)

(def norm (a)
  "normalise a vector i.e. sqrt(a.a)"
  (sqrt (dot-product a a))
)

> (dot-product '(5 4 2) '(1 0 0 ))
5
> (norm '(5 4 2))
6.70820393249937
```

help

You can add documentation to a function as an optional string when its defined. This can then be accessed by typing (help function) and it will return what information there is and what parameter or arguments it expects. It's pretty terse! Here's how to add to your own functions:

```
(def test (x y)
    "This is test"
    ())

> (help test)
(x y) : This is test
LSharp.Function
```

Concluding Remarks

If you have a robobuilder I hope you are inspired to get it out of its box and try out some of the demos. The latest versions of the functions described are available from my (open source) project site (http://code.google.com/p/robobuildervc/).

Why would you use L# to program your robot?

If you want more than an electronic automaton the robot needs "smarts". You either need to carry those around with you by using add on processing boards such as roboard, or you can use your PC remotely, offloading the complex stuff. The main advantage of the remote approach it's cheap - if you already have a PC it cost nothing! Although you might want to buy the bluetooth module so you can be untethered.

How does L# compare with alternatives?

The good: It's very small (80K!!), its open source, its free (I like this), it works, and it works well with .NET. Essentially it's a very powerful scripting language. Plus its lisp features make it good for List processing (surprise!). Is it only for remote PCs? LSharp.net is based on arc syntax and you can run arc on small linux embedded processors. Here's a link to ShevaPlug running Arc on Linux. Of course you loose .NET libraries - and would need to work out the Linux equivalents. http://arcfn.com/2009/08/worlds-smallest-arc-server.html

Are there problems with L#?
- It doesn't support Return from functions, which makes coding somethings harder.
- It also is missing exception handling - you can throw but not catch.
- I've also noticed that an infinite loop seems to crash it which is annoying and needs to be handled.
- There is Very little documentation its syntax is not always clear.
- Its complete lack of trace and debug means getting stuff to work involves putting prn statements in every where

What are the alternatives LISP?
I've not looked at these in any detail. DotLisp looks a lot more fully realised than LSharp. IronScheme uses the new DLR libraries from Microsoft and is huge. GnuCommonLisp is probably very good at Lisp - but not so easy to interface to .NET libraries (I suspect). Here are links if you interested in finding out more:

DotLisp	http://dotlisp.sourceforge.net/dotlisp.htm
GnuCommonLisp	http://www.cs.utexas.edu/~novak/gclwin.html
IronScheme	http://www.codeplex.com/IronScheme

Which text editors to use is a vital question for Lisp programmers as you need parentheses checking built in! Here are the links to the text editors I've looked at. My favourite is Notepad++ at the moment. Its Lisp syntax highlighting and regular expressions make it easy to use.

Here are download links:

Notepad2	http://sourceforge.net/projects/notepad2/
programmers Notepad	http://www.pnotepad.org/download/
Emacs	http://ftp.gnu.org/pub/gnu/emacs/windows/
Notepad++	http://sourceforge.net/projects/notepad-plus/files/

Acknowledgements

- RobobuilderLib uses RBC serial protocol which Robosavvy forum members (I-bot and Raymond) identified.
- Blockworld and Doctor programs were inspired by reading LISP (by PH Winston and BKP Horn): http://people.csail.mit.edu/phw/Books/

Appendix A – Function List

Default functions/variables

Function	Description
- & xs	Subtraction.
* & xs	Returns the product of the xs. (*) returns 1.
/ & xs	Division.
+ & xs	If the first x is a number, returns the sum of xs, otherwise the concatenation of all xs.
< & xs	Less than.
> & xs	Greater than.
and (& xs)	Logical and
apply (f x)	
atom? x	Returns true if x is an atom.
bound (sym)	Returns true if sym is bound in the current environment.
caar seq	Returns the first item of the first item in a sequence.
cadr seq	Returns the second item in a sequence.
car seq	Returns the first item in a sequence.
cddr seq	Returns the rest of the rest of a sequence.
cdr seq	Returns the rest of a sequence.
coerce x t	Converts object x to type t.
compile expr	Compiles the expression and returns executable code.
cons x seq	Creates a new sequence whose head is x and tail is seq.
def (name parms & body)	Defines a new function.
do & body	Executes body forms in order, returns the result of the last body.
do1 & body	Executes body forms in order, returns the result of the first body.
each (x xs & body)	Macro
empty? x	Returns true if x is an empty sequence.
err exception	Raises an exception
eval expr	Evaluates an expression.
even (n)	Returns true if n is even
every? (f xs)	
expt (x y)	Exponent i.e. x to power y
first seq	Returns the first item in a sequence.
for (x start finish & body)	For loop
help (f)	Prints help documentation for f
idfn x	The identity function, returns x.
inspect (x)	Inspects the object x for debugging purposes.
is a b	Returns true if a and b are the same.
isa (x t)	Returns true if x is of type t.
iso (x y)	Isomorphic comparison of x and y.
last seq	Returns the last item in a sequence.
len seq	Returns the length of the sequence.
length seq	Returns the length of the sequence.
let (var val & body)	
list & xs	Creates a list of xs.
load filename	Loads the lsharp expressions from filename.
mac - (name parms & body)	Creates a new macro.
macex (x)	
macex1 (x)	
map (f s)	Maps a function f over a sequence.
member? item seq	Returns true is item is a member of seq.
mod & xs	Returns the remainder when dividing the args.
msec	Returns the current time in milliseconds.
new t & xs	Constructs a new object of type t with constructor arguments xs.
no n	Returns true if n is false, false otherwise.
nor args	Macro
not n	Returns true if n is false, false otherwise.
nth n seq	Returns the nth element in sequence.
null nil false	/empty / false
odd (n)	Returns true if n is odd

or (& xs)	Logical Or
pair (xs (f list))	Split list into pairs
pr (& xs)	Print
prn (& xs)	Print (with newline)
progn & xs	progn xs
range (a b (c 1))	Create range)
reduce (f s)	
reference & xs	Loads the given list of assemblies.
rest seq	Returns the rest of a sequence.
reverse seq	Reverses the sequence.
safeset (var val)	Macro
seq x	Returns x if x is a sequence, otherwise returns a Sequence representation of x.
seq? x	Return true if x is a sequence.
set (x y)	Set a variable
sleep (n)	Sleeps for n seconds
some? (f xs)	
sqrt (n)	Square root of number
stderr	Returns the standard error stream.
stdin	Returns the standard input stream.
stdout	Returns the standard output stream.
str (& xs)	Convert to string
testify (x)	
throw exception	Raises an exception
time (expr)	Macro Times how long it takes to run the expression.
toarray seq	Returns an object[] containing all the members of a sequence.
tolist seq	Returns a list containing all the members of a sequence..
true	
type x	Returns the Type of x.
typeof t	Returns the Type object named t.
uniq	
unless (test & body)	
using xs	
when (test & body)	
while (test & body)	
with (parms & body)	

If using the L# / Lsharp engine within PCremote / Preset form additional variables and functions are available

	Variables initialised at start time
form	Windows form access that LISP is running in
pcr	PCremote object handle
wck	wckMotion object handle
sport	Serial port object handle
	Loaded in at start time from init.lisp[1]
add (x y)	Add two vectors x and y together
alert (x)	Display alert box!
basic18	stand
byte? (x)	check if x byte
checkVer	Check version of robobuilder firmware
dcmodeOff	exit DC mode (amber light off)
dcmodeOn	Enter DC mode (amber light on)
diff (x y)	Subtract two vectors x and y
dot-product (a b)	Calculate dot product of two vectors.
getallServos (n)	Get the current position of attached servos
getServoPos (n)	get position of servo id n
message (x)	Display message
norm (a)	normalise a vector i.e. sqrt(a.a)
number? (x)	check if x number
play (x)	Play motion file

readAcc	read accelerometer values
readDistance	Read distance
readIR	Read IR - simulate
readSn	Read serial number
readVideo	Read Video
remote?	check if robot connected
repeat	repeats
serial?	check if serial port connected
setPassive (n)	set servo id n to read mode
setServoPos (id pos torq)	set servo position (id 0-31) (pos 0-254) (torq 0-3)
setSyncMove (id torq pos)	Synchronous mode
show-doc (x)	Shows doc for an environment entry
smove (a b n tm)	smooth move position a to b in n steps

Appendix B – File List

Location	Contents / description of what it does
http://robobuildervc.googlecode.com/files/LSharp with RBLib.zip	LsharpConsole.exe, Lsharp.dll, RobobuilderLib.dll If you want you can build and download the L# yourself from LSharp.org - there is a svn to extract latest version (v2).
http://robobuildervc.googlecode.com/files/Lisp.zip	See below
http://robobuildervc.googlecode.com/files/final.doc	This document

All the files are available from this zip file. Unzip the file in same directory as Lsharpconsole and then type (load "Lisp\\xxxx.lisp") to load and run.

Filename	How to run	Description of what it does
final.lisp	(run_robobuilder)	You will need a robobuilder and a serial connection for this to work
wckutils18.lisp	(demo)	This will do a punch left - assuming run_robobuilder first
utilities.lisp	(plotaccel 5)	This has lots of utilities but it include the windows display routines: This redefines readAcc to use dummy ransom data - need to reset for actual use - see code.
joy.lisp	(demojoy)	This will wait for joystick input and display what was pressed
km.lisp	(doctor)	The match function and example program doctor.
asdfg.lisp	auto runs	simple blockworld demo
test.csv		punch left as a CSV file
WALK.csv		walk converted to CSV file
maze2.lisp	(maze)	Robobuilder runs around and plots map of maze
grip.lisp		
listen.lisp		Simple voice recognition (uses .NET System.Speech libraries)
speak.lisp		Simple speech synthesis (uses .NET System.Speech libraries)
voicemenu.lisp		Voice control the gripper with simple commands (stand, open, close, exit)