

# Robotic Programming for The Boe-Bot

## Introduction

The Parallax Boe-Bot, or Board of Education Robot is a simple robot made from a Parallax Board of Education (BoE) mounted on an aluminum frame to which two Futaba hobby servos are mounted. These servos have been modified to provide continuous motion and have several speeds at which they will turn. The BoE has a small solderless prototyping board on which several experiments can be done. In order to realize the fascinating potential the BoE-Bot has to demonstrate robotic behaviors, you will need to know how to program it! In the following chapters I will explain how to make the robot move randomly, or purposely, how to seek out or avoid bright light and how to avoid objects. I will also explain how to program your BoE-Bot in such a manner that all of these functions seem to be operating at the same time. Further, I will show you can get behaviors from your robot that you didn't even program in to it!

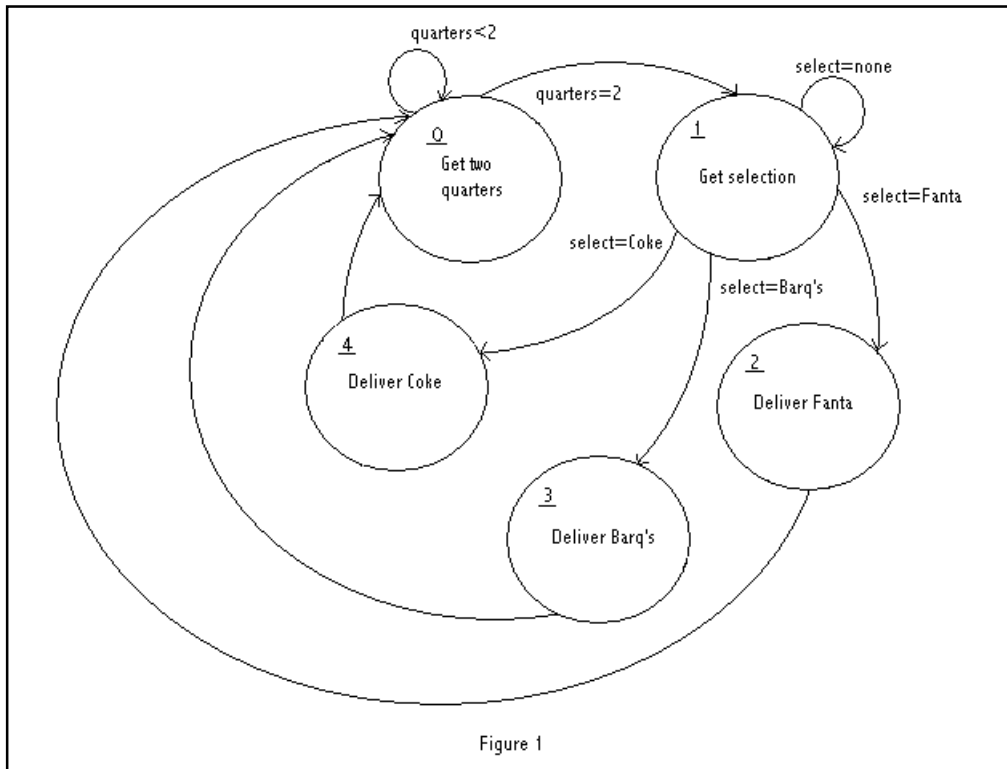
## Linear Execution vs. Concurrent Execution and the Finite State Machine

In a program, each statement is executed in sequential order, the next statement cannot start until the last one is done. When there is only one thread of logic running, this means that the program proceeds in a linear fashion, from start to finish. Concurrent execution is when there is more than one thread of logic running at what appears to be the same time. "How do I do that?", you may ask. In the Parallax Basic Stamp II processor you can do this by using independent sections of code, called *modules* or in our case, *subroutines* that will be called many times before their function is complete. In order to do that, you will need to keep track of where you are in your subroutine so that you know where to start up again when you return. One very good way to do this is called the *Finite State Machine* or *FSM* for short. Because there are only a few different actions you want your subroutine to execute, and it does actually complete its job at some point, it is a *finite* list. There are several forms of FSMs, this type of FSM is a hybrid we will use specifically for robotic programming. This *behavior FSM* is a type of a state machine that returns no outputs, it merely changes state based on input and the current state. Each activity that the FSM engages in, is a *state*, unique in operation and distinct from all other states by its definition.

If this is difficult to understand, lets use a *sort of* real world application to explain FSMs, the soda machine. In order to keep our soda machine simple, we will create a pretty stupid machine with these abilities:

- takes only quarters
- needs two quarters to get a soda
- will not give you your money back
- does not give change, nor return money that isn't a quarter
- has Coke, Barq's Root Beer and Fanta Orange soda
- has an infinite amount of soda and never runs out

As you can see, we have eliminated all of the *exceptions* or error conditions that a normal soda machine could see in order to simplify this explanation - its artificial for a reason; we're not designing soda machines. However, do pay attention to exceptions and error conditions when you are designing your own FSMs! Figure 1 shows a graphical rendering of our soda machine FSM. The underlined numbers in each of the circles are the state number for that state. A line with an arrow denotes a transition from one state to another (the arrow points the direction.) If a transition line is labeled, that label is the result of the transition function and defines the condition required for that change of state. An unlabeled line is a transition that will always occur as soon as the function of that state is completed. The lines that loop back upon a state show iteration, or that the FSM remains in this state doing something until a terminal condition is reached, at which time a defined transition, that is labeled will occur. Here we see that our soda machine FSM will remain in state 0 until two quarters have been given, at which point our FSM will transition to state 1. Here we will wait, looking at buttons until a selection is made. When a selection is made, our FSM will then transition to state 2, 3 or 4 depending on the selection made. From these terminal states, our FSM will immediately transition back to state 0 after completing. This is the general process of definition and representation for the FSMs that we will be using to define our BoE-Bot behaviors.



Remembering where you are in your subroutine is called *saving state* and is essential if you are to pick up where you left off when last this subroutine ran. Each state in our behavior FSM will be executed when its subroutine is called and will exit the subroutine when that state is completed. Subsequent calls of that subroutine will execute the next correct state that is defined. Why is this useful? Lets look at two code snippets that show why this can make your whole program run faster. Both of these pieces of code operate the hobby servos that make your BoE robot move, don't worry about understanding them exactly, what this code does will be fully explained in a later chapter. The one thing you must know is that a hobby servo requires that a pulse of 1ms (millisecond) to 2ms must be sent to each servo every 20 to 30ms or the servo will not perform correctly. If you send it too often (say every 7ms) the servo will jitter, if you send it too rarely (say every 50ms) then the servo will stop. These pulses need to be repeated continuously, and regularly in order to operate correctly, a single pulse is not very useful to a servo.

Linear based servo controller subroutine	FSM based servo controller subroutine
<pre> act:   for I = 1 to 10     pulsout LEFT,750     Pulsout RIGHT,750     pause 20   next return </pre>	<pre> act:   if aDur &gt; 0 then aDec     aDur = 5     pulsout LEFT,750     pulsout RIGHT,750     goto aDone aDec:   aDur = aDur - 1 aDone: return </pre>

The code on the left looks very simple and fast, but looks can be deceiving. The *pulsout* instructions are used to output a pulse of the needed width to turn the servos. Remember, this pulse needs to be repeated every 20 to 30 milliseconds (ms) in order for the servo to respond properly. Also, it needs to have several repetitions of this pulse for the motor to turn and keep running. The *pause* instruction will cause the Stamp II to pause for 20ms, each of the pulses sent will be 2 microseconds \* 750, or 1.5ms. So, each pass through

this *for/next* loop will take  $3\text{ms} + 20\text{ms} = 23\text{ms}$  at least, 10 times through the loop will take 230ms! That is almost 1/4 of a second when nothing else can be done!

Now lets look at the code on the right that implements a two state FSM to move the servos. You can see that our subroutine on the right does one of two operations at any given time. The first operation is to output the pulses to the servos and set the *aDur* variable. The second operation is to simply decrement the *aDur* variable. In either case, after the operation has been accomplished we exit the subroutine. Each of these operations will be defined as a *state* for the *act* behavior.

We will get into more details on how to describe and design state machines for our robotic behaviors using examples and programs that you will write for your BoE-Bot in later chapters.

Returning to our code samples, lets figure the time spent in the subroutine on the left now. Since the Stamp II executes about 4000 lines of code a second this means that each instruction will take about 250us to execute. The pulsout instructions will obviously take 1.5ms each to execute because that is the length of the pulse that is being sent. In state 1 it will take 3ms for the pulsout instructions + 750us for the other three instructions, which equals 3.75ms. In state two our second subroutine will take about 750us of processor time each time it is executed.

Instead of the 230ms of processor time taken by the left subroutine, we now will take  $5 * 750\text{us} + 3.75\text{ms} = 7.5\text{ms}$  of processor time total (we are taking 5 turns through it after the initial pulse outputs remember?) to accomplish the same purpose. If we only count a single 23ms loop for each pass through the first subroutine, we will have saved 15.5ms of processor time, which, at 4000 instructions per second amounts to 62 instructions that can be executed elsewhere and give us the exact same activity on our servo motors. If we take into account the full 230ms time for the left loop we save over 226ms which is a whopping 904 instructions!

But why is this important? A robot does not just wander aimless around in its environment, it usually has some task to accomplish. Whether it is searching for a fire to put out, trash to pick up or for another robot to attack, it is doing something else more important than just running its motors. When we use the motor driver routine on the left above, the robot is doing absolutely nothing but concentrating on running the motors for 230ms. During this time it cannot look at a sensor, pick up trash or put out a fire. If it runs into something, it will just keep running into it until it is finished with that loop and can then do something else. Each of the other behaviors that we implement in our robot will be some activity that the robot will need to perform in a timely manner. It does us no good to detect an object to avoid *after* we have already run into it! Let us assume that our robot is running the following behaviors, listed in lowest priority to highest priority, to achieve some objective:

- Go North until home is found (chooses a direction to travel)
- Avoid hitting anything by using IR proximity detection (if something is a danger, choose another direction)
- If I hit something, back up and turn left (chooses *yet another* direction to go)
- Stop and beep when I am home (choose no direction at all, just stop)
- Select the highest priority direction to go and call *act* to implement it

Many of these behaviors will tell the motors to perform some action; backing up, turning left, whatever. Each of these behaviors will need to reference sensors in order to perform their actions. Each of these behaviors (using the system I am suggesting) will be Finite State Machines implemented in subroutines that will be called from within some main code loop (you will see some of these behaviors defined later on.) In the motor driver routine *act* shown above as the right side code snippet there is a variable *aDur* defined. When the *act* FSM is first called *aDur* is set to 5. This means that the *pulsout* instructions will be executed once, then the next 5 times the *act* subroutine is called it will do nothing but decrement *aDur* and exit. This means that it will spend as little time in the subroutine as possible. Why is this useful? It is useful because *act* only needs to send those *pulsout* actions once every 20 to 30ms. Our robot can be looking at sensors and selecting the next motor action while it is waiting to send that next series of pulses out. In this way, we *use* the time it takes to read sensors and make decisions in those other four subroutines as the delay we *must* take between pulses we send to the servo motors instead of *wasting* that time with a *pause* instruction! In effect, this makes it look like everything is happening at the same time instead of one thing after the other. If we were to use a linear programming model instead of Finite State Machines to implement all of

our behaviors and actions then we would not be able to look at the compass or check for an obstructing object until all 230ms in the code snippet on the left had completed. In that time our robot might miss seeing the chair in front of it and collide with it before it gets a chance to change direction. There is nothing special about the number 5 chosen for *aDur* either, I used that number as a suggested starting point. In reality this number is chosen by trial-and-error to achieve the smoothest timing. I started with this number in my own robot and as I added behaviors I reduced it. For example, with four behaviors active I have my *act* routine set *aDur* to only 2.

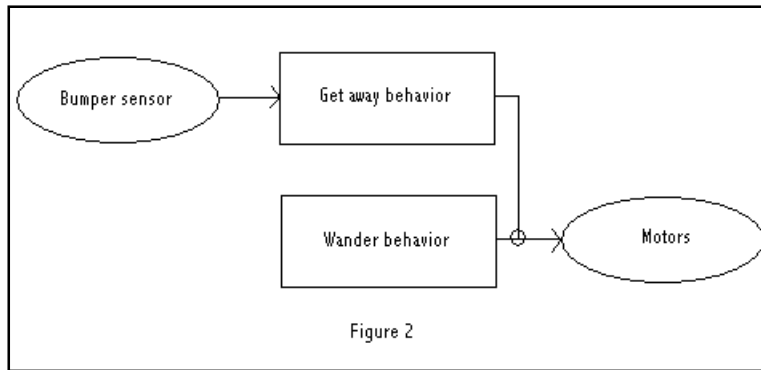
When we implement all of our behaviors as FSMs this has the effect of interleaving the code that needs to be executed in each subroutine so that no one behavior needs to wait until the prior behavior completes in order to do at least some of the work that needs to be done in its own routine. This improves our robot's response time to its environment. In the case of the *act* subroutine above, this results in a smoother motor response and quicker reaction to obstacles and objectives.

Before using the Finite State Machine method of behavior implementation I would notice that my robot would appear to hesitate longer and longer as I added more and more complex behaviors. This FSM method will all but eliminate this hesitation. It is more complex to design and code than linear programming, but the results, I feel, merit the complexity. Try programming your robots both linearly and using FSMs, I think you will agree that using Finite State Machines improves your robot's abilities and allows us to get as much out of our Stamp II as we can! Eventually a set of behaviors may become so complex that even using FSMs will not prevent some hesitation, but we can do much more using FSMs as our programming model rather than linear programming before that happens.

## Subsumption Architecture and Robotic Behavior

To subsume a task is to supercede it with a higher priority task. When we speak of subsumption in robotic programming we are describing the process by which one behavior subsumes, or over-rides another based on an explicit priority that we have defined. This behavioral architecture was first described by Dr. Rodney Brooks in his article "A robust layered control system for a mobile robot" in *IEEE Journal of Robotics and Automation.*, RA-2, April, 14-23, 1986. The *Brooksian* ideal subsumption architecture does not require that any behavior know anything about any other behavior. A robot programmed in this manner responds reflexively to its environment using simple behaviors. What this also means is that new behaviors can be added to a robot without changing *any other behavior*. This makes enhancing a robot's programming very simple. An example of subsumption programming in our little BoE-Bot would be, for instance, if we had a behavior that simply randomly chooses a direction to travel and a duration of time to travel in that direction. When complete, that subroutine (behavior now) would choose a new direction and duration. But we don't want to get stuck up against a wall or table leg, so we add a behavior that looks at a bumper and if we run into something will make our robot back up and turn away. We want the *bumper* behavior to have priority over the *wander* behavior, so it will subsume that behavior and take over the control of the wheel motors to get itself away from the wall. Because the *bumper* behavior has a higher priority than the *wander* behavior we can be assured that our little BoE-Bot can get away from obstacles that block its path. Further, if *wander* had set a vary long duration on the last direction that it wanted to go, when *bumper* was done, the *wander* behavior would take up the control again, continuing on its merry way as if nothing had happened. Because each behavior is independent, and many simply react to the robot's environment, we can build up a set of behaviors whose interactions with each other are *not* programmed, and we will begin to see combinations of behaviors unforeseen! This is called emergent behavior because we didn't plan it, it came about because of the robot's interaction with its environment. When emergent behavior is allowed to occur our robots appear to take on a life of their own and stop acting as if they just do the same thing over and over again. More importantly in the *real* world, some behaviors can be programmed to do a certain task, or retrieve an object, and not have to be concerned with the moment-to-moment job of avoiding falling into a hole or running into a rock.

Our simple sets of behaviors we just discussed can be graphically displayed by a subsumption network diagram. An example of our simple wander/bumper behaviors is shown in Figure 2 below. The ovals on the left are inputs, the rectangles in the middle are behaviors and the ovals on the right are outputs.



Where a line from a higher priority behavior intersects a line from a lower priority behavior to its output actuator (in this case, Motors) the higher priority behavior is said to have *subsumed* control from the lower priority behavior. This is shown on the network above by *Get away's* line intersecting *Wander's* line with a circle around the junction. There can be multiple intersections in multiple locations on the direction lines that can show the subsumption logic very clearly. Feel free to innovate how you show your network for your robot, there are many ways to build a subsumption network diagram.

### Where do we go from here?

Get used to these diagrams and ways of thinking. In the chapters ahead we will be using these ideas and expanding on the simple diagrams that we have seen so far. In each section I will break our work down into easy to see pieces, such as variables needed, I/O ports used and new instructions. I will also show the process by which the state machines are defined and our subsumption network diagrams can be used to understand our behavioral priorities.

## **Chapter 1: Making the BoE-Bot Move, and the Finite State Machine**

A robot that doesn't do anything isn't very interesting. Now that you have modified your servos, attached them to Stamp II I/O ports and have checked their operation, we will make our BoE-Bot wander randomly around its environment. To implement this simple functionality we need to have two modules: One that determines a direction and a duration to go in that direction and another to send commands to our wheels. We will call these modules *wander* and *act* respectively. *Act* simply operates the motors. It isn't really a behavior, its an output and will be labeled on our diagram as *motors*. *Wander* is a behavior so it will appear on our subsumption diagrams as a behavior.

### The State Machine and Stamp II Code for *Act*

We know that to get our servos to turn the wheels we need to output a pulse whose width needs to be approximately 1ms to 2ms long, with 1.5ms being the stop position. We also know that this pulse needs to be repeated every 20ms to 30ms for our servos to continue moving. This suggests that we will have two operations that we will need to program. One will be "output the pulse", the other will be "wait for 20ms" after which we go back to state 1. We have two servos, so we could say that we have three operations, output left servo pulse, output right servo pulse and wait. To keep this module simple we will limit it to two states, one to output the pulses and one to delay before the next pulse time. Since we know that we need to repeat the pulse every 20ms to 30ms and we know that we don't want to spend all of our time in the *act* module waiting for this time to elapse, we therefore know that we will be exiting and entering this block of code several times before the full set of state transitions occurs. We know that each instruction takes about 250us to execute for the Stamp II, this tells us about how long each module that we will be calling will take to run, we count up the instructions and multiply by 250us. For now we'll guess and say that it will take 5 passes through state 2 for 20ms to elapse. As we write other modules we will be revising this estimate, but

this is a good starting point for now. Lets graph our state machine so that we can understand what we want, and maybe even see any mistakes we made in our logic!

Before we can build our state machines it is helpful to define all of the actions that we wish to have our behavior or output function perform. Each of these functions can be grouped by their activities into an individual state. Here is a way to define a motor control function, which we saw above as the subroutine *act*.

State 0

- Output left servo pulse value
- Output right servo pulse value
- Set the number of iterations (aDur) = 5

State 1

- Decrement aDur
- If aDur = 0 then go to state 0

We now have a detailed list of actions that need to occur in our state machine for *act*. We can now draw a state diagram that has the needed detail, such as Figure 3 below.

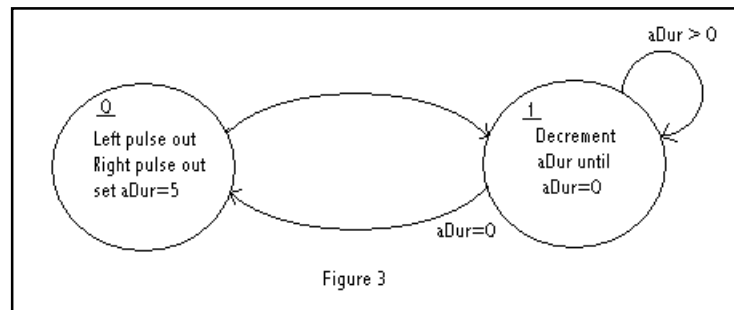


Figure 3

This state diagram tells us when each transition is taken and exactly what each state is doing. The transition from state 0 (on the left) and state 1 always occurs, there is no condition. However, the transition from state 1 (on the right) to state 0 occurs when aDur = 0. Notice that state 1 has a loop that exits and turns back on itself. This indicates that this state *iterates* on itself, in our case, this means that we will enter this part of the module several times before it is complete. Look at this diagram carefully. It shows us very clearly what our *act* module must do, and in what order it must do it. Think about how it would look if we were to output the left motor pulse in a different state than the right motor pulse, how would that look? For our purposes, we can say that each circle (state) is a place where we enter the module in our program and decide what to do next. You should now be starting to see how powerful this concept is for us!

Here is the code that implements our *act* module. I will show the variable declarations on the left and the actual code on the right from now on. To make a program easier to read and to modify, you should make liberal use of *con* statements and not use "magic numbers" embedded in your code. This is how I will show our programs. Remember that everything that follows a ' is a comment, not an instruction. I will explain why we multiply the lmotor and rmotor values by 10 in the next chapter.

<pre> 'Servo routine cons and vars LEFT  con 15      'port 15,left motor RIGHT con 3       'port 3,right motor SACT  con 5       'times through the loop drive  var word   'both sides in var ldrive var drive.byte1 'left side is here rdrive var drive.byte0 'right side here aDur  var byte    'duration counter </pre>	<pre> act:'servo controller subroutine   if aDur &gt; 0 then aDec     aDur = SACT          'do state 1     pulsout LEFT,lmotor * 10     pulsout RIGHT,rmotor * 10     goto aDone          'state 1 done aDec:   aDur = aDur - 1      'do state 2  aDone: return </pre>
--	--

If we are to write a large program we need to conserve variable space, the PBASIC language has some very clever ways to use that variable space. The *pulsout* instructions output a pulse to our servo motors of

the proper width for the speed and direction that we want. The *drive* variable is a word which is two bytes, a *byte0* and a *byte1*. You will see why this is a useful way to represent the motor drive variables when we design the code for the *wander* module in the next chapter!

## Chapter 2: Making the BoE-Bot Wander

All we need *wander* to do is randomly choose a direction and a duration to go that direction. This module will be as easy to design as the *act* module, and it will demonstrate the ease with which we can add new behaviors, and how these behaviors can remember what they are doing. The first step in designing our form of a finite state machine is to write down all of the steps that are to be taken and all of the transition functions that need to be tested. After we have our list, the state machine can be drawn.

Here is our action list for *wander*:

State 0

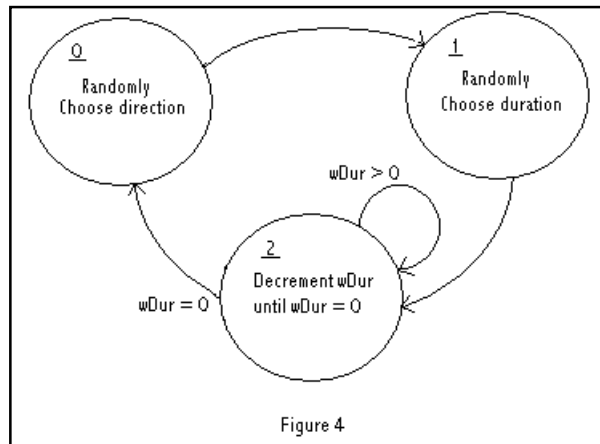
Choose a direction to go by using the *random* function (*wDir*)  
Go to State 1

State 1

Choose a duration to go by using the *random* function (*wDur*)  
Add some minimum time to the duration so its not too short  
Go to state 2

State 2

Decrement *wDur*  
If *wDur* = 0 then go to state 0



The state machine diagram for our *wander* behavior is shown in figure 4 below.

Some of the arrows do not have transition labels attached to them. When a state machine automatically transitions from one state to another and does not need to make a decision a transition function is not needed.

The purpose of the *wander* module is to randomly choose a direction for our robot to go and a random time duration for it to take going there. The *random()* instruction plays an important role in the *wander* logic. *Random* uses a *word* variable type and needs a "seed" to set up its return value. We will just use the last return value it gave us as the seed for the next one. We will also use a *mask* to only get numbers in a certain range; a small range for the direction changes, a much larger one for the duration. Another interesting feature is the *lookup* instruction. This instruction uses a *word* sized variable, because we break the *drive* variable into a *byte0* and *byte1* variables, we can get both the left and right motor values from a single lookup! You'll see how this is done in the following code, and you'll see how this makes it very simple for us to change what the robot will do.

'Servo drive commands	wander :
fd      con      \$6432      'forward	branch wstate,[wDir,wDur]

rv	con	\$3264	'reverse	'This is state 2
st	con	\$4b4b	'stop	wDur = wDur - 1
tr	con	\$644b	'turn right	if wDur > 0 then wDone1
tl	con	\$4b32	'turn left	drive = wDir 'get direction
rr	con	\$6464	'rotate right	wstate = 0 'reset state
rl	con	\$3232	'rotate left	wDone1: 'completed
'wander values				return
wstate	var	byte	'FSM status	wcDir: 'choose direction
wDir	var	word	'wander value	random seed 'random direction
wDur	var	byte	'wander duration	i = seed & %111 'mask off for 0-7
				lookup i, [tr,fd,fd,fd,rr,fd,fd,tl],wDir
				'choose direction
				wstate = 1 'next state
				return
				wcDur: 'choose duration
				random seed 'randomize
				wDur = (seed & %111111) + 20
				'mask for 64 and add 20 for more time
				wstate = 2 'next state
				return

On the left all of the useful motor control commands are listed with a \$ in front of them. This means that those numbers are hexadecimal, I have used hexadecimal (HEX) numbers because its easy to see the left and right halves of them (byte1 and byte0) that we have assigned as the left and right motor values. Of course, this means that we need to understand HEX in order to know what is going on. In HEX, each digit represents a number from 0 to 15, A, B, C, D, E, and F represent the numbers 10, 11, 12, 13, 14, 15 respectively. The digit to the far right is the '1's digit, just like in our decimal system, but it can be from 1 to 15. The next digit to the left is the '16's digit, multiply any number you see there by 16, then add the number in the '1's digit place to that to get the decimal number. Our numbers above are 4 digits, but remember, that is just because we are putting both the left and right motor values in at once, we can look at the two left digits as separate numbers from the two digits to the right. This means that the HEX number \$1C = 16+12, or 28. Remember in the last section when I said I'd explain why *act* multiplies the *ldrive* and *rdrive* number by 10? Now is the time for that explanation. A byte can only hold values from 0 to 255, our servos need to be sent numbers from 500 to 1000. To get that range, we use 50-100 (which are numbers less than 255) for each motor speed setting, then multiply by 10 to get values from 500 to 1000. This doesn't represent all possible values, but this really doesn't matter when driving modified servo motors as wheels.

We now have almost all we need to make our robot wander aimlessly through a room. Almost. Our *wander* and *act* functions are subroutines, this means that something has to call them. The loop defined by *main* and *goto main* is our "brain" for our robot. All behaviors are called from here. This is the full program that has all of the variables and the subroutines as well as the setup and main run loop to show you how it all fits together. Type this into the Stamp II programmer and watch your little robot wander around the room for a while. Make sure it doesn't run into anything! We'll teach it to avoid running into the wall in a later chapter.

```
'Generic values
i      var    byte    'loop counter, whatever
tmp    var    word    'temporary holder
seed   var    word    'random number seed

'These are for the servo routines
LEFT   con    15      'left wheel port
RIGHT  con    3       'right wheel port
SACT   con    5       'times through act routine
drive  var    word    'wheel command combo
ldrive var    drive.byte1 'left wheel command
rdrive var    drive.byte0 'right wheel command
aDur   var    byte    'duration of pulse left

'Servo drive commands
fd     con    $6432   'forward
rv     con    $3264   'reverse
st     con    $4b4b   'stop
tr     con    $644b   'turn right
tl     con    $4b32   'turn left
rr     con    $6464   'rotate right
rl     con    $3232   'rotate left
```

```

'wander values
wstate var    byte           'shared byte
wDir   var    word           'wander value
wDur   var    byte           'wander duration

'set up for running
wstate =0                'initial wander state

main:                    'this is the main activity loop
  gosub wander
  gosub act
goto main

'=====
'Behaviors follow
'=====
wander:                  'randomly wander around
  branch wstate,[wcDir,wcDur] 'state 2 immed. follows
  wDur = wDur - 1
  if wDur > 0 then wDone1
    drive = wDir           'get direction
    wstate = 0            'reset state
wDone1:                  'completed
  return
wcDir:                   'choose direction
  random seed             'random direction
  i = seed & %111         'mask off for 0-7 only
  lookup i,[tr,fd,fd,fd,rr,fd,fd,tl],wDir 'chose direction
  wstate = 1             'next state
  return
wcDur:                   'choose duration
  random seed             'random direction and duration
  wDur = (seed & %111111) + 20 'mask for 64 choices
  wstate = 2            'next state
  return

act:                      'moves servo motors
  if aDur > 0 then aDec   'already doing one, got here
  aDur = SACT             'times through this one
  pulsout LEFT,ldrive * 10
  pulsout RIGHT,rdrive * 10
aDec:                     'decrement stuff
  aDur = aDur - 1
aDone:
return

```

Note the *set up for running* section in the code. We need to make sure that the *wander* behavior starts out in the correct state, so, we set that state here. Now, we need to make sure that *wander* and *act* are called regularly, so these two subroutine calls are placed in a loop starting at *main*. This loop will be a very important feature of our code when we start adding new behaviors to our robot!

### Chapter 3: The Photophobic Robot and Subsumption Programming

It would be interesting if the robot had some higher purpose than just wandering randomly around the room. Lets make a cricket-like behavior, a bug that looks for a dark corner in which to hide. We describe the behavior that makes our robot seek darkness and avoid light as *photophobic* which means "fear of light". Fear is a living being's emotion, but we can make our robot *appear* to fear light with this behavior! Build the light sensor circuits in experiment #3 to use this code. To read the photocells we will use the Stamp II instruction *rctime*. Reading a photocell takes time, the darker it is, the longer it takes to read the cell. Because it takes so long to read a single photocell, we will want to break up the readings into different states, doing the math to determine whether or not the left or the right photocell sees brighter light also takes time. We don't want our robot to stop and think every time it looks at the light readings to the left and right, so again, these will be separate states. When we have made a decision, we want to turn in the direction of the darker reading. We should turn for a while, so we will be setting a duration for the turn as well as a direction. If we don't have a certain duration, our robot could easily jerk back-and-forth, which doesn't look like its being very decisive! This will be a more complex state machine than our previous ones because it has five states instead of two. Here is a list of actions that will need to be done:

State 0  
 Read light level on left photocell  
 Set state = 1

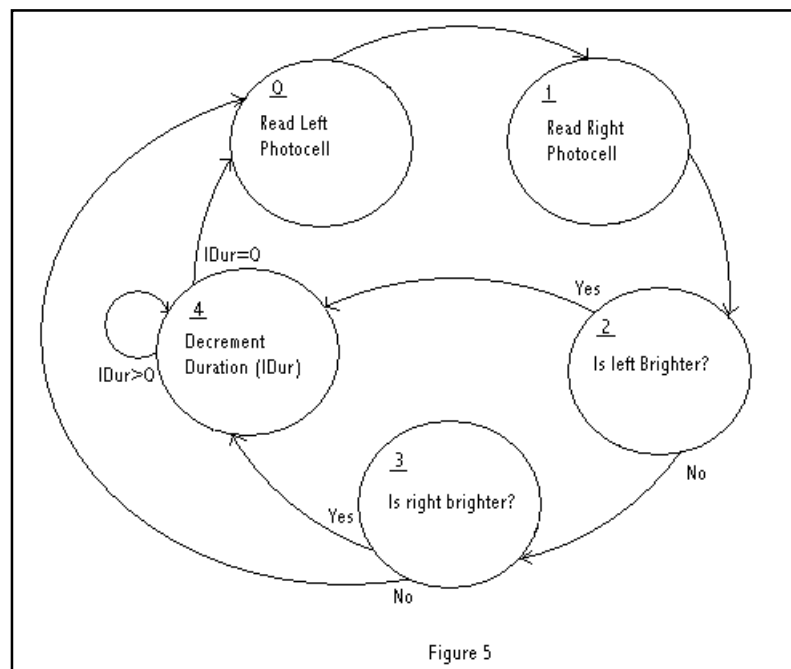
State 1  
 Read light level on right photocell  
 Modify this value by 1.5 because this cell reads a little lower than the other one  
 Set state = 2

State 2  
 Add margin to left reading  
 If left is brighter than the right  
 Set IDir = turn right (tr)  
 Set IDur = 30  
 Set lstate = 4 (next state is decrement)  
 Else  
 Set lstate = 3

State 3  
 Add margin to right reading  
 If right is brighter than the left  
 Set IDir = turn left (tl)  
 Set IDur = 30  
 Set lstate = 4  
 Else  
 Set lstate = 0 (do nothing, both sides are about the same)

State 4  
 Decrement IDur, IDur = IDur - 1  
 If IDur = 0  
 Set lstate = 0 (start over from beginning, we are done)  
 Else  
 Do nothing, come back to state 4 again for a decrement, we are still turning

Figure 5 shows the state machine we will be using for our *photophobic* behavior.



This state machine was drawn only after all of the actions that were required had been written down and organized in a logical manner. Because we are changing variables and passing data between these states, we really should include that in the transition function arguments (the captions on the arrows). Sometimes it is easier to create the state diagram and build the functionality of each state from that graph. There is more than one way to build a behavior, this is just one suggested design.

Notice that the description of the states is pretty detailed. Your descriptions should be this detailed so that you don't forget anything that could be important. In State 1 a "fudge" has been added to the reading. This is because no two photocells are the same, when these were measured, one read a little lower than the other, this modification evened out the photocells for comparison. This is something for you to check with your photocells too. Of course, since our state machine is quite a bit more complex, you can expect the code to be more complex as well. It is, but if you write the steps that need to be taken very carefully, in a programming-like language (as seen above), you can practically write the software straight from your state description. This "almost" programming language is called *pseudo code*, it helps you to organize your thoughts logically. If you can't write it down, you can't program it! Here is the actual Stamp II code for this behavior:

<pre>'light looker vars and constants LLIGHT con    11      'left sensor RLIGHT con    4       'right sensor pleft  var    word    'left value pright var    word    'right value lstate var    byte    'FSM state lDur   var    byte    'how long to go lDir   var    word    'where to go LMARG  con    15      'light margin</pre>	<pre>lightlook:   low LLIGHT      'set up for sensors   low RLIGHT      'branch takes 200us    branch lstate,[lread1,lread2,lcomp1,lcomp2]   lDur = lDur - 1  'state 4 decr duration   drive = lDir    'correct direction   if lDur &gt; 0 then lDone1 'still counting   lstate = 0      'restart FSM  lDone1:   return          'done  lread1:   rctime LLIGHT,0,pleft 'get left value   lstate = 1        'go next state   return  lread2:   rctime RLIGHT,0,pright 'get right value   tmp = pright &gt;&gt; 1     'compensation   pright = pright + tmp   lstate = 2          'go next state   return  lcomp1:   tmp = pleft +LMARG   'set threshold   if tmp &gt; pright then lDone2   'left not past threshold   lDir = tr           'bright to left   lDur = 30          'for a while   lstate = 4         'go decr state   return  lDone2:   lstate = 3        '2nd compare   return  lcomp2:   tmp = pright +LMARG 'set threshold   if tmp &gt; pleft then lDone3 'not past   lDir = tl         'bright to right   lDur = 30        'for a while   lstate = 4       'go decr state   return  lDone3:   lstate = 0       'none past   return</pre>
---	---

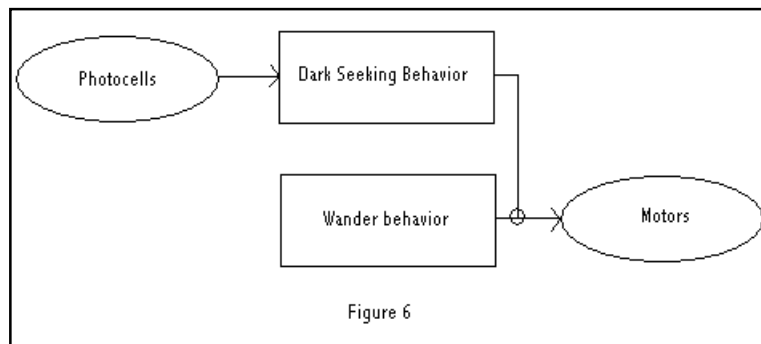
There are a few subtle short cuts in this code. The `tmp = pright >> 1` statement divides `pright` by two and stores the result in `tmp`, then the next line adds `tmp` to `pright`. This multiplies `pright` by 1.5 to compensate for its lower readings than the left photocell. The branch statement is a way to branch to a different section of code based on an index value, in our case that index value is our state.

Add the variable and constant column on the left to the top of your program. Add the *lightlook* behavior subroutine to the end of your program. Add this behavior to the *main* programming loop like this:

```
'set up for running
wstate =0                'initial wander state
lstate =0                'initial photophobic state

main:
  gosub wander
  gosub lightlook
  gosub act
goto main
```

What do you think will happen? The *wander* behavior will set the variable *drive* to some random direction, however, the *lightlook* behavior, knowing nothing at all about *wander* will then set *drive* to some other direction perhaps, **if** it sees a darker corner to run to. Finally, *act* will use the *drive* value to determine which motors to drive in which direction. So, the *lightlook* behavior will have a higher priority than the *wander* behavior because it gets the opportunity to set the *drive* variable **after** *wander* does. Are you starting to see the exciting possibilities with this type of robotic programming? Lets look at the subsumption network diagram in Figure 6 to see the behaviors we have just given our BoE-Bot.



Here is a challenge for you! Turn our little photophobic robot into a cricket that hides in the dark and chirps. You can start with this list of requirements:

- Both photosensors must read some low level of light (high reading = low light)
- When the low light threshold has been reached, the robot must stop there
- Use the last readings taken from *lightlook*, this means you won't need to take new readings
- The robot will only chirp when it is standing still in the dark
- This will be higher priority than the *lightlook* or *wander* behaviors
- If either sensor's value goes above the threshold, then the robot will move again

By using the last read light values that were taken by the *lightlook* behavior we are violating the strict rule of independent modules. However, in the interest of saving the time that these readings would take, this is a small infraction. Besides which, this isn't a traffic law, and very little side-affect occurs from this minor violation. Feel free to add the *rctime* functionality if you wish.

Now your task will be to create a detailed list of actions that must occur in each state of this finite state machine. Then draw a state diagram, decide where this behavior belongs in your robot's subsumption network and finally write the behavior code and insert the *gosub* for the behavior into the *main* loop. Make sure you initialize your state machine in the *Set up* section before the main behavior loop!

## Chapter 4: Don't Run Into the Walls, The Avoid Behavior

We have random movement and photophobic behaviors programmed into our robot now. However, BoE-Bot does not have any way to avoid running into things. A behavior that will avoid walls and furniture will certainly extend the usefulness of our little robot! Build the IR proximity detectors (IRPD) from

Experiment #4. Lets think about what we need to do to avoid collisions with objects. An enhancement to this behavior would be to have the robot turn 180 degrees when it sees an obstruction directly in front of it. How would you go about creating this behavior?

```

Get IRPD readings
If first reading then
  Save it
Else
  If this reading = last reading then
    Choose direction and set drive
  Clear reading history
Else
  Save this reading

```

This one isn't as complex as the *lightlook* behavior; going around something is a simpler behavior than actively seeking something (dark) is. Some things are so simple that no state machine really needs to be built. One could say that this behavior really does have two states, the first just takes an IRPD reading, the second takes another and sees if they agree. However, these two actions are so closely coupled that it would be difficult, and unnecessary to separate them out.

Here is our code for *avoid*:

<pre> 'IRPD vars and constants ileft var in9 'IR LED outputs iright var in0 'i=(see code) IEN con 5 'enable for 555 ilast var byte 'hit counter </pre>	<pre> avoid:'IRPD routine High IEN 'enable 555 i=0 i = ileft * 2 + iright 'read IRPD low IEN 'disable 555 if ilast = I then ickit 'two reads agree goto iDone 'just first read ickit: 'This line chooses new direction lookup i,[rr,tr,tl,drive],tmp drive = tmp i=0 'clear history iDone: ilast = i 'new history return </pre>
--	---

Math in the Stamp II is evaluated from left to right, so you either must be careful of your order of operations or use explicit parentheses to force the correct order. The IR demodulators that we are using detect a signal by sending a 'low' or '0' back on that I/O port line. Therefore, with our code above, a 3 means no detection, a 2 means detection on the right, a 1 is a detection on the left and a 0 is detection on both lines. Our *lookup* instruction includes the old *drive* value in its list, this is done so that we can change nothing if there is no detection, and not modify the *drive* value at all. The *avoid* module will continue to change our robot's direction until there is no obstacle detected. This will look like a smooth search until the path is clear!

To add this behavior, place the variables and constants block near the top of your current program and put the *avoid* subroutine in your behaviors code section. Since we want *avoid* to have higher priority than any other behavior we have done so far, place it in the *main* loop as shown below:

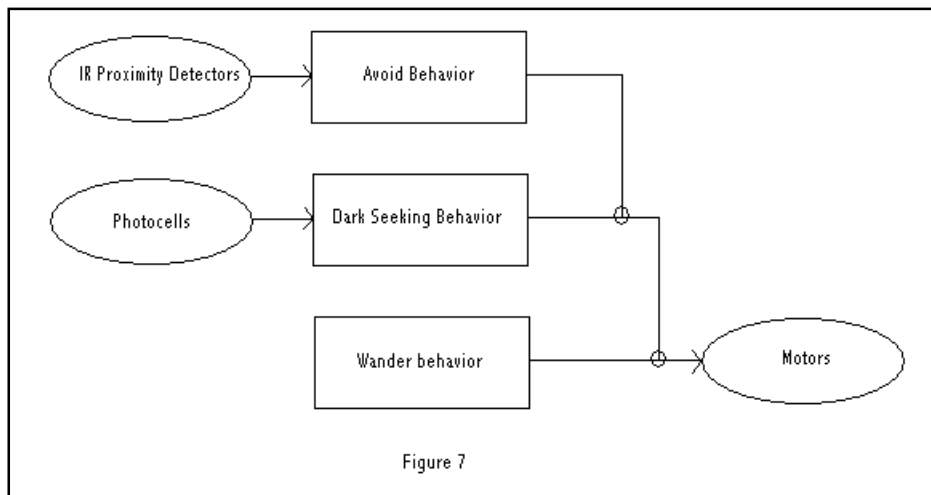
```

'set up for running
wstate =0 'initial wander state
lstate =0 'initial photophobic state
ilast =0 'initial avoid history value

main:
  gosub wander
  gosub lightlook
  gosub avoid
  gosub act
goto main

```

Figure 7 shows our subsumption network diagram with all of our behaviors included.



Subsumption networks are quite useful to explain the potential activity of our robots to others. They can be used to predict behavior or to compare behaviors of other robots with our own. Reading these diagrams can be easier than studying other peoples' programs! This type of programming can be used in any application that needs good response to external stimuli that is not dependent on precise time intervals.

Here is another challenge for you and your BoE-Bot projects. Add a behavior that will cause your robot to backup and turnaround when it bumps into an object. This will require you to build a bumper and connect it to an I/O port on the Stamp II first. Then you will need to decide on a list of actions that need performed, break these actions into states and decide what you will need to remember. You will also need to decide what priority a bumper reaction should have in your subsumption network. You now have a set of tools for programming these behaviors into your BoE-Bot, go have fun!

### ***Future Projects for the BoE-Bot***

There are potential plans in the works for future articles and projects for the BoE-Bot. Here is a partial list of what is being considered:

- Direction finding with a compass
- Communication between two BoE-Bots for cooperative activities
- Motion sensing
- SONAR range finding