

# Advanced Robotics with the Toddler

---

**Student Guide**

VERSION 1.3

PARALLAX 

## **WARRANTY**

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

## **14-DAY MONEY BACK GUARANTEE**

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

## **COPYRIGHTS AND TRADEMARKS**

This documentation is copyright 2002-2004 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following Conditions of Duplication: Parallax Inc. grants the user a conditional right to download, duplicate, and distribute this text without Parallax's permission. This right is based on the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

This text is available in printed format from Parallax Inc. Because we print the text in volume, the consumer price is often less than typical retail duplication charges.

BASIC Stamp, Stamps in Class, Board of Education, SumoBot, and SX-Key are registered trademarks of Parallax, Inc. If you decide to use registered trademarks of Parallax Inc. on your web page or in printed material, you must state that "(registered trademark) is a registered trademark of Parallax Inc." upon the first appearance of the trademark name in each printed document or web page. Boe-Bot, HomeWork Board, Parallax, the Parallax logo, and Toddler are trademarks of Parallax Inc. If you decide to use trademarks of Parallax Inc. on your web page or in printed material, you must state that "(trademark) is a trademark of Parallax Inc.", "upon the first appearance of the trademark name in each printed document or web page. Other brand and product names are trademarks or registered trademarks of their respective holders.

**ISBN 1-928982-14-X**

## DISCLAIMER OF LIABILITY

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

## INTERNET DISCUSSION LISTS

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from [www.parallax.com](http://www.parallax.com) via the Support → Discussion Groups menu. These are the forums that we operate from our web site:

- [BASIC Stamps](#) – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- [Stamps in Class<sup>®</sup>](#) – Created for educators and students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- [Parallax Educators](#) – Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a forum for educators to develop and obtain Teacher's Guides.
- [Parallax Translators](#) – The purpose of this list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications.
- [Toddler Robot](#) – A customer created this discussion list to discuss applications and programming of the Parallax Toddler robot.
- [SX Chips and Tools](#) – Discussion of programming the SX microcontroller with Parallax assembly language tools and 3rd party BASIC and C compilers.
- [Javelin Stamp](#) – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java<sup>®</sup> programming language.

## ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to [editor@parallax.com](mailto:editor@parallax.com). We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, [www.parallax.com](http://www.parallax.com). Please check the individual product page's free downloads for an errata file.



## Table of Contents

<b>Preface</b> .....	<b>v</b>
Why learn to Walk?.....	v
Educational Concepts from the Toddler .....	v
Audience and Teacher’s Guide .....	vi
Foreign Translations .....	vi
Toddler Discussion Group .....	vi
Special Contributors .....	vii
Parts Required for Chapters #8 and #9 .....	vii
<b>Chapter #1: Assembling the Toddler Robot</b> .....	<b>1</b>
Meet the Newest Member of the Family .....	1
Let’s Build the Toddler .....	2
Tools Required .....	2
Step #1: Install Top Tilt Servo in Body .....	3
Step #2: Install Stride Servo in Body Bottom .....	4
Step #3: Electrically Center Servos.....	5
Step #4: Install Servo Horns .....	8
Step #5: Install Brass Wire Keepers on Stride Servo.....	9
Step #6: Install Top Plate .....	10
Step #7: Install Body Rods for Legs.....	11
Step #8: Slide Legs onto Toddler Body.....	12
Step #9: Secure Legs to Toddler Body .....	13
Step #10: Assemble Stride Linkages .....	14
Step #11: Attach Ankles .....	16
Step #12: Attach Feet .....	17
Step #13: Install Ball Joints .....	18
Step #14: Install Tilting Rods .....	20
Step #15: Install Battery Box.....	22
Step #16: Install Standoffs .....	24
Step #17: Connect Toddler Board to Battery Pack .....	25
Step #18: Install 4 AA Batteries .....	26
Step #19: Mount Toddler Board .....	27
Step #20: Center Servos Again and Make Adjustments .....	28
<b>Chapter #2: Taking Your First Steps</b> .....	<b>29</b>
Simple Control and Your First Program.....	29
Servo Control Basics .....	29
How a Servo Works .....	29
Time Measurement and Voltage Levels.....	30
Many Ways to Move the Toddler .....	32

Approach #1: Brute Force .....	32
Approach #2: Data Tables .....	32
Approach #3: State Transitions.....	33
Theory of Operation.....	33
Activity #1: Basic Walking Movements: Brute Force .....	36
Activity #2: Walking Backwards: Brute Force .....	45
Activity #3: Using a DATA Table to Store Movements .....	50
Activity #4: Using State Transitions For Movements .....	56
Challenges.....	64
<b>Chapter #3: Turning Around .....</b>	<b>65</b>
Slip Sliding Turns.....	65
Activity #1: Making a Turn .....	65
Activity #2: Different Turns .....	71
Challenges.....	72
<b>Chapter #4: Coordinated Walking .....</b>	<b>73</b>
Multiple Tables for Many Actions.....	73
Activity #1: Which Table? .....	73
Activity #2: Figure 8s and Square Dancing .....	79
Challenges.....	88
<b>Chapter #5: Following Light.....</b>	<b>89</b>
Bringing Your Creature to Life .....	89
Activity #1: Building and Testing Photosensitive Eyes .....	90
Programming to Measure the Resistance.....	92
How the Photoresistor Display Works.....	95
Activity #2: A Light Compass.....	96
Programming the Toddler to Point at the Light .....	96
How the Light Compass Works.....	102
Activity #3: Following The Light.....	104
How the Light Follower Program Works .....	110
Challenges.....	111
<b>Chapter #6: Object Avoidance with Infrared .....</b>	<b>113</b>
Wireless Object Detection .....	113
Infrared Headlights for Toddler .....	114
The FREQOUT Trick .....	115
Activity #1: Building and Testing the New IR Transmitter/Detector .....	117
Testing the IR Pairs .....	118
How the IR Pairs Display Program Works .....	120
Activity #2: Object Detection and Avoidance .....	121
Sampling Between Movements .....	121
Challenges.....	123

<b>Chapter #7: Staying on the Table</b> .....	<b>123</b>
What's a Frequency Sweep? .....	123
Activity #1: Testing the Frequency Sweep.....	123
Programming the IR Distance Gauge .....	123
Activity #2: The Drop Off Detector .....	123
Aliased Variables .....	123
Activity #3: Toddler Shadow Walker .....	123
Programming the Toddler Shadow Walker .....	123
Challenges.....	123
<b>Chapter #8: Real-Time Programming</b> .....	<b>123</b>
What is Meant by Real-Time .....	123
A Processor's Got To Know Its Own Limitations.....	123
Where's Real Time in Toddler?.....	123
Activity #1: Building and Testing Toddler Toes and Infrared Circuit.....	123
Activity #2: Finite State Machine Real-Time Walker .....	123
Finite State Machine .....	123
Commented Out DEBUGs .....	123
State Indicator LEDs .....	123
Do Vision.....	123
Challenges.....	123
<b>Chapter #9: Acceleration-Based Motion</b> .....	<b>123</b>
Measuring Toddler's Incline.....	123
More On Real-Time Techniques .....	123
Activity #1: Building and Testing the Tilt Sensor.....	123
Noise Filter .....	123
Activity #2: Integrated Sensor Walking Program .....	123
What's the Problem? .....	123
Challenges.....	123
<b>Appendix A: Parts Listing and Sources</b> .....	<b>123</b>
<b>Appendix B: Toddler PCB Schematic</b> .....	<b>123</b>
<b>Appendix C: Blue Bell Designs Co-Processor</b> .....	<b>123</b>
<b>Index</b> .....	<b>123</b>





# Preface

---

## WHY LEARN TO WALK?

Walking robots are relatively unexplored in the area of hobby and education. The reason for this is that they have inherent design limitations such as their ability to handle diverse terrain including stairs; they require more complex programming algorithms and sensor-based feedback; and interdependency between software and hardware requires some critical thinking to obtain the desired actions.

This may be one of the reasons most of our robots have wheels. However, people have a natural tendency to appreciate a walking robot: they seem more like human beings; they offer more entertainment value because they're fun to watch; and to make a biped robot walk successfully is a challenge worthy enough to pursue the concept. If you succeed with the Parallax Toddler™ robot we'd consider you a well-trained roboticist.

Aside from programming, walkers have real applications on difficult terrain. Robots with multiple legs and multiple degrees of freedom ultimately handle more complex terrain.

The Toddler robot simplifies the walking robot concept. While the Toddler certainly won't be caring for the elderly, bringing you a drink from the kitchen, vacuuming the house or driving you to the store, it will provide a first exposure to the concept of a programmable biped. Through this text you'll discover the complexities and rewards associated with learning to program a walking robot. Walking robots introduce embedded control in a positive, fun and friendly way.

## EDUCATIONAL CONCEPTS FROM THE TODDLER

Educators ask Parallax what they will learn from our different curriculum. The Toddler is considered an advanced robotics project and introduces the following concepts:

- Dependencies between mechanical and electrical systems and the ability to tune hardware or adjust software to obtain desired results
- Advanced programming skills with the BASIC Stamp 2. An efficient Toddler program makes use of Stamp programming tricks with the DATA statement and EEPROM pointers, program routines that are reused and “configured” prior to execution, variable aliasing, general sound programming practices (constant/variable

definitions that allow for program customization in just a few places rather than throughout an entire program)

- Advanced robotic programming techniques including real-time programming, subsumption architecture and state machines.
- A step-wise process which starts with the basics and builds to something more complex and ultimately more useful

If you need help, call or e-mail Parallax for technical support. We'd be pleased to help get your Toddler walking the way you want it to.

## **AUDIENCE AND TEACHERS GUIDE**

Students as young as 14 years old should be able to build and program the Parallax Toddler. Because of the Toddler's more extensive mechanical system and more complex programming we believe that the youngest student to have success with this kit would probably be about 12 years old. If you have experience otherwise please let us know at [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com). The Advanced Robotics with the Toddler text presently has no teacher's guide. Based on demand we may elect to produce the answers to challenge questions posed in this text.

## **FOREIGN TRANSLATIONS**

Parallax educational texts may be translated to other languages with our permission (e-mail [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com)). If you plan on doing any translations please contact us so we can provide the correctly-formatted MS Word documents, images, etc. We also maintain a discussion group for Parallax translators which you may join from [www.parallax.com](http://www.parallax.com) via the Support → Discussion Groups menu. This will ensure that you are kept current on our frequent text revisions.

## **TODDLER DISCUSSION GROUP**

Parallax customer Mike G. Otis started the Toddler Robot discussion group. You can subscribe to this group for free from [www.parallax.com](http://www.parallax.com) via the Support → Discussion Groups menu. Discussion includes different bipedal robots and how they operate, customizing to the Toddler with different sensor and hardware, and sharing of new PBASIC code. Many pictures, files and interesting links have been posted on this discussion group.

## **SPECIAL CONTRIBUTORS**

Parallax designed the Toddler mechanics, yet we recognize that the first time we have seen the two-servo concept employed for a walking robot is United Kingdom robotics designer David Buckley's "Big Foot". Though the core concept of a two-servo walker now seems simple, we recognize that Mr. Buckley created the ingenious use of two servos for a walker. Big Foot is a plywood kit available through Milford Instruments ([www.milinst.com](http://www.milinst.com)) of the United Kingdom. David Buckley endorses the Parallax Toddler robot and contributed to the Toddler design as part of our team.

This text was jointly authored by Parallax, Inc., Harry Lewis ([www.bluebelldesign.com](http://www.bluebelldesign.com)) and Bill Wong of Pennsylvania. Harry makes a BASIC Stamp<sup>®</sup> co-processor for robotics which is featured in the appendix. Bill is an editor with Electronic Design magazine and a serious BASIC Stamp robotics enthusiast. His daughter has won County, State and Intel Science Fair awards with her maze solving robotic projects.

## **PARTS REQUIRED FOR CHAPTERS #8 AND #9**

The Toddler Robot Kit (#27310 or #27311) you have received includes all the components for Chapters 1 through 7. Chapters 8 and 9 require additional parts you may purchase from Parallax. Though we aim to make the kit as complete as possible, including the additional parts would have simply raised the price of the Toddler kit and reduced the number of users who could purchase the robot. Here is what you will need for Chapters 8 and 9, should you decide to do the final experiments:

- Chapter #8: Toddler Bumper Switch Toes (Parallax #27312)
- Chapter #9: Memsic 2125 Accelerometer (Parallax #28017)

The last two chapters demonstrate some of the most interesting Toddler projects, such as state machine programming, subsumption architecture and real-time sensor feedback.



# Chapter #1: Assembling the Toddler Robot

---

## MEET THE NEWEST MEMBER OF THE FAMILY

No matter how easy it looks, you'll soon realize that the mechanical movements and BASIC Stamp<sup>®</sup> code required to make a two-servo biped move in a distinct fashion is more complex than its rolling counterpart, the Parallax Boe-Bot<sup>™</sup> robot. In fact, the Toddler's finished construction uses more sensors for feedback. This includes infrared emitters/detectors for objects seen at a distance; bumper switches ("Toddler Toes") should infrared not "see" the object, and an accelerometer for determining the tilt parameters.

The Toddler is capable of doing everything a rolling robot can do if you've got the patience to tune the mechanics and software. Not only is the robot more entertaining than a rolling robot, you'll become a more proficient programmer as you learn to exploit the BASIC Stamp's capabilities. The Toddler will show you how to design a PBASIC program that uses constants and variables, program pointers and EEPROM access for data storage. A well-designed program means you can easily tune the software for the right mechanical control in just a few places rather than rewriting your entire program.

The Toddler's motion is from two servo motors (the type normally used in remote controlled airplanes). The Toddler's top servo motor is used to rotate the robot's center of gravity back and forth over the two feet, and the bottom motor moves both legs back and forth. The legs use a simple parallel linkage to the ankles that keeps the feet parallel to the ground. Both legs are linked together through the same motor so that as one leg move forward, the other moves backwards.

By controlling one motor at a time the robot can move forward, backward, and turn either left or right. By blending the control of both motors the robot can do move in a more coordinated fashion with smooth movements.

A surface-mounted BASIC Stamp 2 module controls the Toddler's servo motors and sensors. The BASIC Stamp 2 is used throughout the Stamps in Class educational series and provides plenty of program space, speed and memory for use with a Toddler.

## LET'S BUILD THE TODDLER

The Toddler may be assembled a number of ways depending on the surface on which the robot walks, the type of additions you make with sensors and hardware and the speed which you program the robot to walk. The default assembly method is appropriate for hard, level surfaces and will be used to demonstrate code throughout this text.

### Tools Required

You will need the following tools to build the Toddler:

- Screwdriver (included in kit)
- 5/32" L-key wrench (included in kit)
- Small pliers



**Parts are most easily identified** by looking in Appendix A where we have a complete parts listing, and by looking at the parts picture on the last page of the text.

**Parts missing from your kit** will be replaced by Parallax free of charge. We have a QA/QC program but occasionally we make mistakes. If we left a part out of your kit contact us immediately. If you break parts or want additional hardware for your customized Toddler you can order any piece of this kit from our Component Shop ([www.parallax.com](http://www.parallax.com) under "Products/Component Shop").

**Step #1: Install Top “Tilt” Servo in Body**

The following parts are required for this step:

- (4) 4/40 3/8” long pan-head machine screws
- (4) 4/40 nuts
- (1) Toddler Body
- (1) Toddler Servo Motor

Remove the servo horns from both servo splines by loosening the black screw. You will need these again shortly so put them aside.

Install the servo in the Toddler Body with the shaft oriented towards the bottom of the body as shown Figure 1-1. Position the servo squarely. Using four (4) 4/40 3/8” pan-head machine screws and (4) 4/40 nuts, screw the tilt servo into the body. The easiest way to do this is to hold the nut with one finger while turning the screwdriver with the other hand.



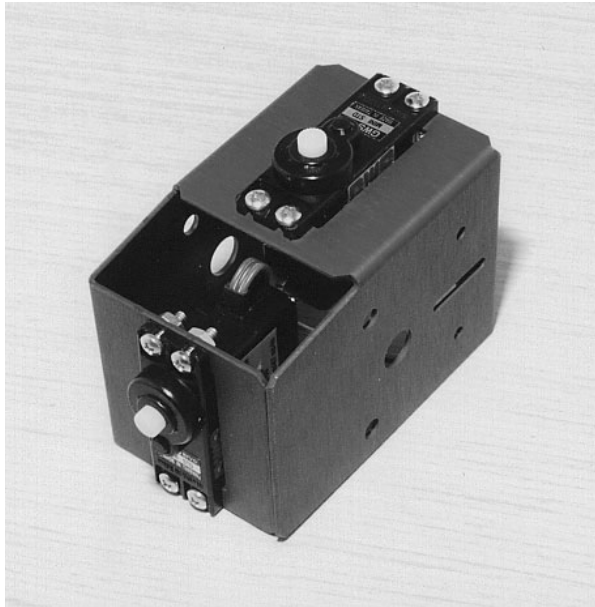
**Figure 1-1**  
Install Tilt Servo in Body

**Step #2: Install “Stride Servo in Body Bottom**

The following parts are required for this step:

- (4) 4/40 3/8” long pan-head machine screws
- (4) 4/40 nuts
- (1) Toddler Servo Motor

Install the bottom servo with the shaft oriented towards the front of the robot as shown in Figure 1-2. Using four (4) 4/40 3/8” machine screws and (4) 4/40 nuts, screw the stride servo into the body.



**Figure 1-2**  
Install Stride Servo in  
Body



### **Step #3: Electrically Center Servos**

The following parts are required for this step:

- Battery pack
- (4) AA Batteries (not included)
- Serial Cable (RS-232 DB-9 and straight-through)
- BASIC Stamp software installed and running on your computer

The servos should be “centered” prior to any further Toddler assembly. This will ease any fine-tuning adjustments by allowing them to be made only in software. Don’t skip this step – it will make future adjustments easier.

Plug the two servos into the Toddler printed circuit board “X6 and X7” connectors. The “B” label on the header connects to the black servo lead. Next, place a jumper wire between the Vs1 and Vss above the breadboard. Place another jumper wire between Vs2 and Vss on the same header. This connects the servo grounds to the BASIC Stamp’s ground as shown in Figure 1-3

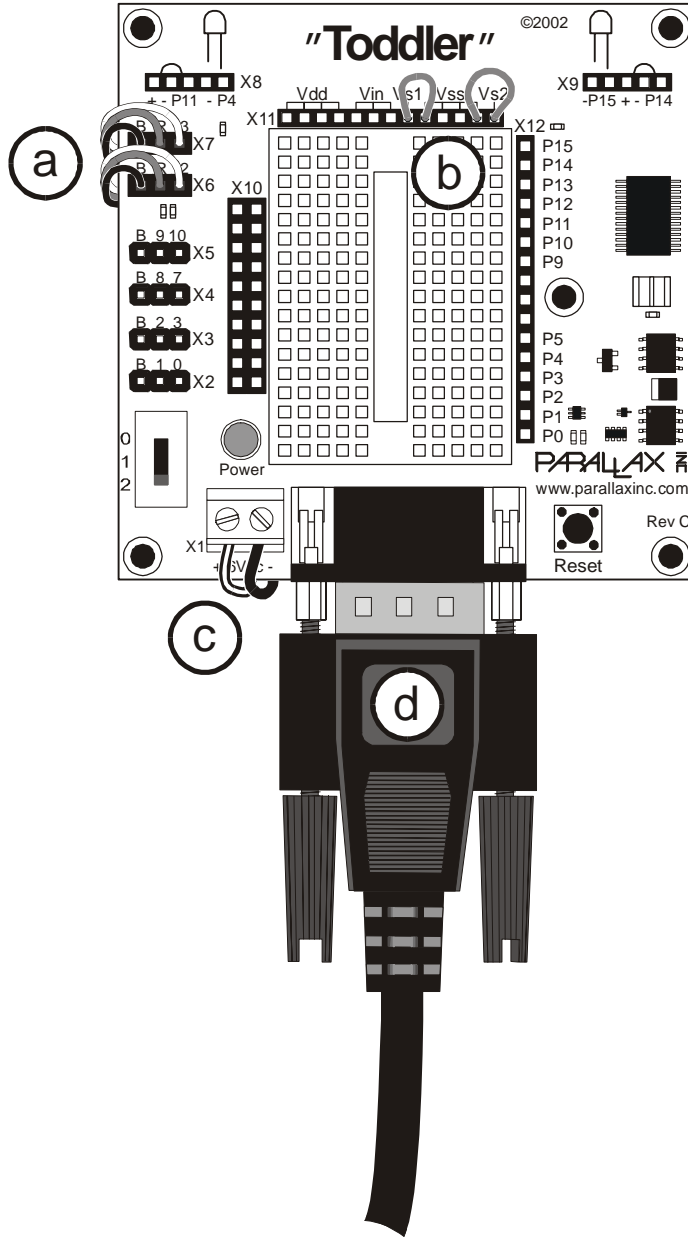


**Why are the servo grounds Vs1 and Vs2 connected to Vss?** The Toddler printed circuit board is identical to another Parallax robot controller: the SumoBot. With the SumoBot, it is possible to be pushing forward while your competitor is actually backing you off the ring. Monitoring each servo’s ground connection with an A/D converter lets you identify this situation and execute an escape movement.

Next, install four batteries in the battery pack. Put the power switch in Position 0. The battery pack’s white wire lead connects to the Toddler board’s “+” terminal block and the black wire lead connects to the “-” terminal block. Use a screwdriver to connect these wires.

Using the serial cable, connect the Toddler board to the serial port of your computer.

.



**Figure 1-3**  
Toddler Servo Centering Setup

- a. Plug servos into the X6 and X7 connector.
- b. Connect Vs1 to Vss and Vs2 to Vss.
- c. Install batteries and connect battery pack to the X1 jack.
- d. Plug in serial cable and connect to PC.

The Toddler has a three-position power switch. The state of each position is shown below. The three-position switch has a middle position that powers the entire circuit except servos. A complete schematic of the Toddler is included in Appendix B.

Position 0 – No Power

Position 1 – Power to everything except servos

Position 2 – Everything is powered

Place the power switch in Position 2. The next step is to program the BASIC Stamp.



**The three-position power switch** will prove to be very useful in future chapters. You will be able to “debug” your Toddler’s sensors without having your robot walk away. This will let you isolate the robot’s subsystems.

Open the BASIC Stamp Windows editor. Write or load the following piece of code that will center both servos:

```
' -----[ Title ]-----
' Toddler Program 1.1 - Servo Centering.bs2
' Electrically center Toddler servos during robot construction
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Declarations ]-----

TiltServo          CON    13    ' Tilting servo on P13
StrideServo        CON    12    ' Stride servo on P12

' -----[ Main Routine ]-----

Center:              ' Center servos with 1500 us pulses
  PULSOUT TiltServo, 750
  PULSOUT StrideServo, 750
  PAUSE 25           ' Wait 25 ms
GOTO Center         ' Repeat
```



**Toddler source code is available for download** from [www.parallax.com](http://www.parallax.com). In our other Stamps in Class texts the code is usually typed by the student, but the listings in advanced Toddler programs are quite long; loading them will save time.

Download your code using the Run/Run menu or by pressing the ► button on the toolbar. This program runs in an endless loop. When the servos stop moving (this will happen almost instantaneously) move the power switch to off (position 0). Disconnect the servos from the Toddler board. Remove the batteries from the battery box and disconnect the leads from the Toddler's screw terminal.

#### **Step #4: Install Servo Horns**

The following parts are required for this step:

- Two straight servo horns
- (2) Small black screws to hold horn to servo

The servo horns should be installed as straight as the spline notches allow without turning the motor from its centered position (Figure 1-4). Secure each servo horn with the small black Phillips head screw using a screwdriver. The screws were removed from the servo spline in Steps 1 and 2.



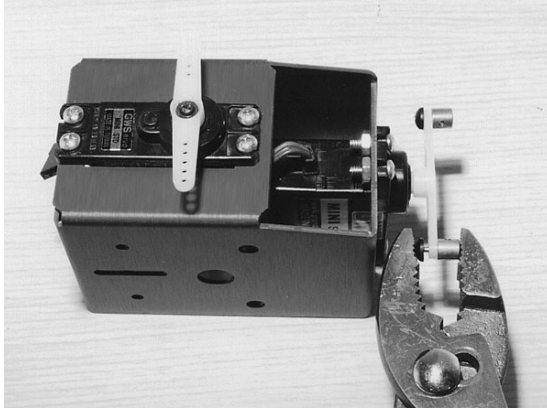
**Figure 1-4**  
Both servo horns are installed on centered servos.

**Step #5: Install Brass Wire Keepers on Stride Servo**

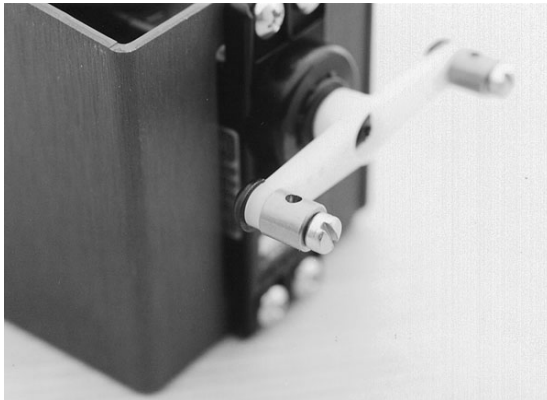
The following parts are required for this step:

- (2) Brass wire keepers (brass holder, set screw and holding grommet – also called E-Z connectors)

Attach the two brass wire keepers on the outermost holes of the stride servo's control horn (the white two-arm plastic piece attached to the servo in Figure 1-5). Using pliers, press the rubber "keeper" onto the post of the brass wire keeper. Put the two small screws into the threads of the brass wire keeper so they don't get lost (Figure 1-6)



**Figure 1-5**  
Install brass wire keepers  
on the stride servo only.



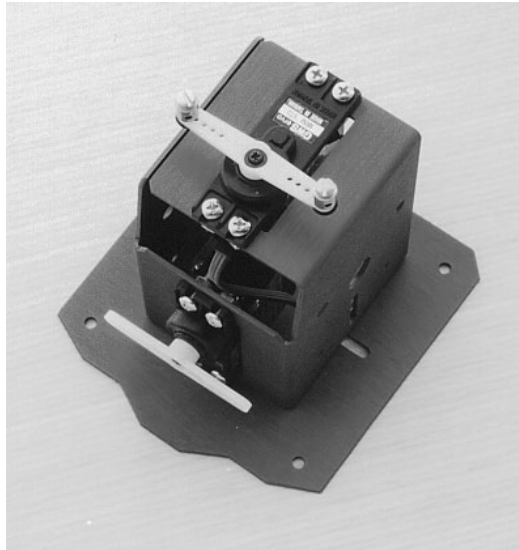
**Figure 1-6**  
Put the two set screws  
back into the brass wire  
keeper so they don't get  
lost.

### **Step #6: Install Top Plate**

The following parts are required for this step:

- Toddler Top Plate
- (4) 4/40 3/8" flathead machine screws
- (4) 4/40 nuts

The top plate is most easily installed by turning the Toddler body upside down as shown in Figure 1-7. Position a 4/40 nut over the hole and insert a 4/40 3/8" flathead screw through the top plate from the bottom. Hold the nut with one hand and tighten the screw with the other hand. Repeat process for three more holes. Note: This step uses the "flathead" countersunk 4/40 screws, not the flat "panhead" screws with the round top.



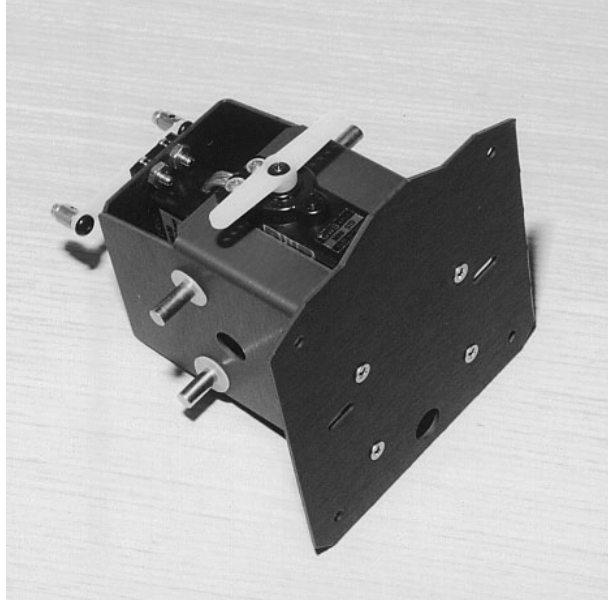
**Figure 1-7**  
Install the Top Plate with  
the flathead screws.

**Step #7: Install Body Rods for Legs**

The following parts are required for this step:

- (2) 3/16" 3" long stainless steel rods
- (4) plastic washers

Slide the two 3" 3/16" stainless steel rods through the two holes in the Toddler's body (Figure 1-8). Slide a plastic washer over each rod. This washer will keep the Toddler's legs from rubbing against the body.



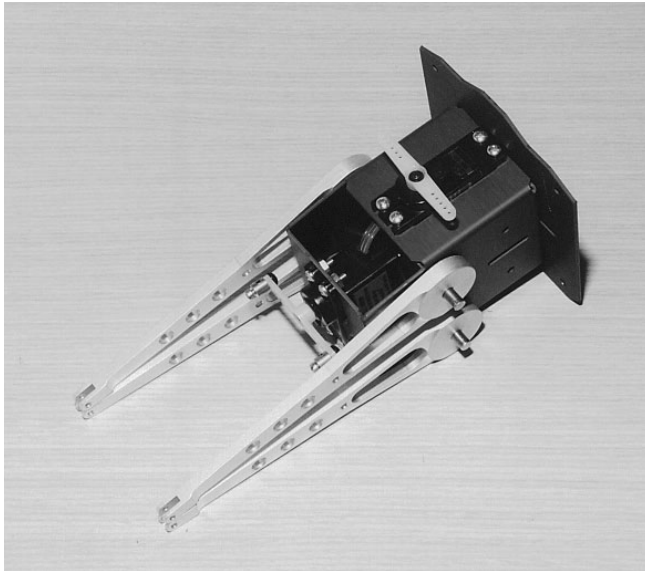
**Figure 1-8**  
Slide two rods into body  
and install four plastic  
washers.

**Step #8: Slide Legs onto Toddler Body**

The following parts are required for this step:

- (4) Toddler legs

Slide the four Toddler legs onto the ends of the stainless steel rods going through the body (Figure 1-9).



**Figure 1-9**  
Slide four legs onto body

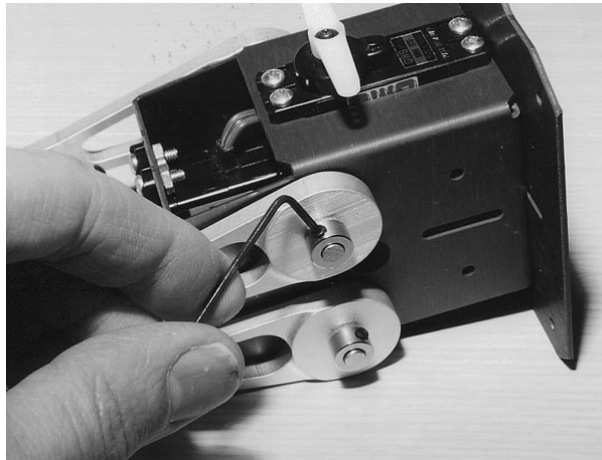


**Step #9: Secure Legs with Collars**

The following parts are required for this step:

- (4) 3/16" collars with setscrew
- L hex-key wrench in collar package

Find the package of 3/16" metal collars and L-key. Slide the collars onto the stainless steel rod. Tighten each collar using the L-key wrench as shown in Figure 1-10. If the setscrew doesn't seem to tighten, slightly angle the wrench to prevent stripping of the set screw or wrench. The legs should move freely against the body without binding. If the legs are too tight against the body the servos will strain and the Toddler won't walk well.



**Figure 1-10**  
Use the 3/16" metal collars. Be sure legs move fairly freely against the body.

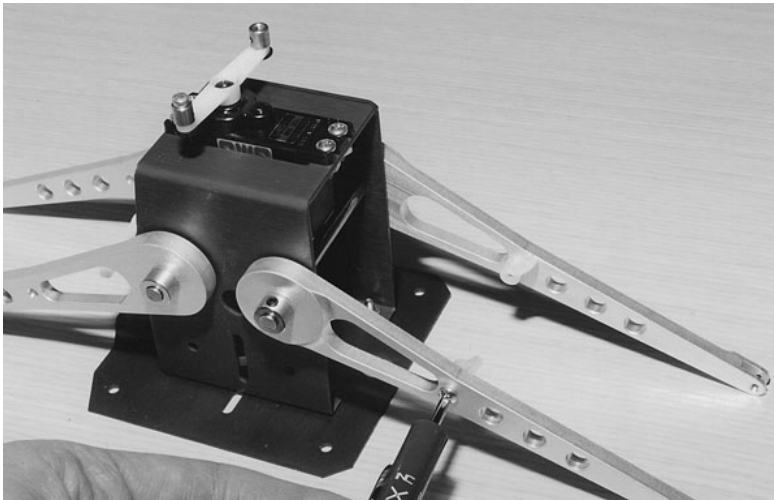
### **Step #10: Assemble Stride Linkages**

The following parts are required for this step:

- (2) 4/40 plastic wire keepers (also called E-Z adjust horn brackets)
- (2) 3/8" 4/40 panhead machine screws
- (2) brass right-angle wires

This is a two step process.

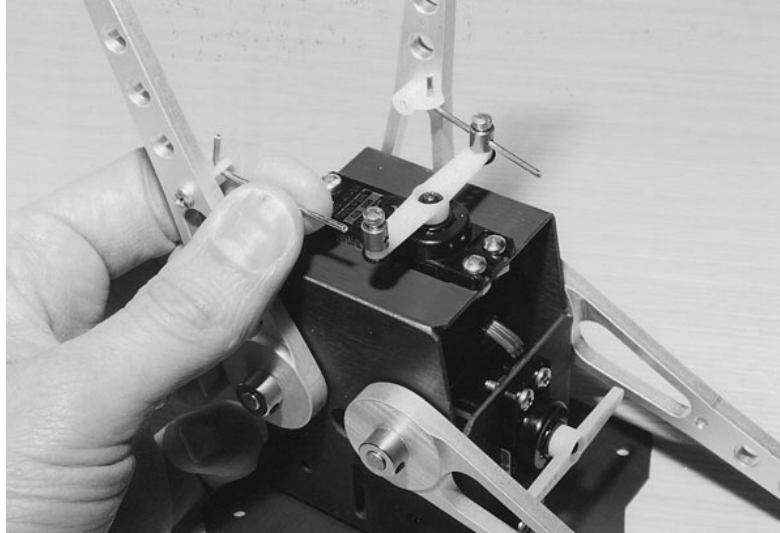
First, insert a 3/8" 4/40 panhead screw through the holes on the rear left leg Figure 1-11. Tighten the screw into the plastic right-angle bracket. Repeat for the process for the rear right leg. Leave the screws loose enough so the bracket and screw can rotate on the leg. Rotate the body 180 degrees.



**Figure 1-11**  
Attach the 4/40  
plastic wire  
keepers with a  
3/8" 4/40 screw.

Using the two right-angle brass wires, slide the straight end through the brass wire keeper hole Figure 1-12. Insert the short end through the top of the plastic right-angle bracket. Tighten the screw to hold the wire.

Adjust the linkages so the legs are vertical, not slanted to either side. Electrically center the servos again if necessary if the servo was accidentally moved. Repeat for the other rear side.



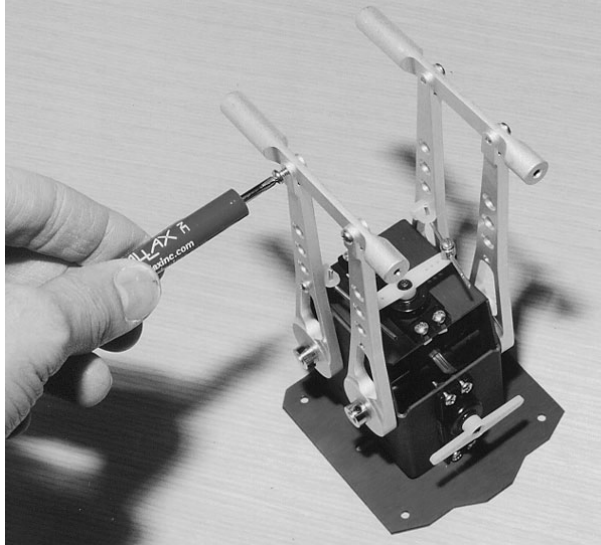
**Figure 1-12**  
Slide four legs  
onto body.

### **Step #11: Attach Ankles**

The following parts are required for this step:

- (2) ankle parts
- (4) 4/40 1/4" panhead machine screws

Attach the ankles to the legs using four 4/40 1/4" screws (Figure 1-13). The longer part of the ankle should be oriented towards the back of the Toddler's body. This placement moves the weight of the robot forward and provides better overall control.



**Figure 1-13**  
Attach ankles.

**Step #12: Attach Feet**

The following parts are required for this step:

- Toddler left foot
- Toddler right foot
- (4) 4/40 stainless steel allen-cap screws
- 3/32" hex L-key wrench

Line up the ankle into the foot's 3rd hole from the instep (Figure 1-14). If it is too tight slightly bend the tabs of the feet outward.

Attach the left ankle to the left foot using two plastic 4/40 screws and the 3/32" L-key wrench. These screws have a round head that acts as a bearing surface on the robot's foot. Repeat for the right side. Ensure free tilting of each foot before proceeding to the next step.



**Figure 1-14**  
Attach feet with Allen-cap screws.

### **Step #13: Install Ball Joints**

The following parts are required for this step:

- (4) ball joints (ball joint with post, nut)

Install a ball joint on the outermost hole of each foot (Figure 1-15). Securing the nut may require a small wrench to tighten the ball joint unless you have a pair of needle nose pliers. One way to do this is to hold the nut with a finger and turn the ball joint until tight. Don't grip the ball with pliers as this can damage it and prevent it turning freely in its plastic socket when assembled.



**Figure 1-15**  
Install ball joints on feet.

Install two ball joints on the tilting servo control horn (Figure 1-16). Use the outermost two holes for these ball joints.



**Figure 1-16**  
Install ball joints on servo horn.

### **Step #14: Install Tilting Rods**

The following parts are required for this step:

- (2) .090" diameter 5.4" stainless steel rod, 2/56 thread on each end
- (4) ball joint plastic cups with 2/56 thread

Thread two plastic ball joint cups onto the ends of the 5.4" stainless steel rod. Place the rod next to the control horn and foot for sizing. To properly position the ball joints make the finished piece about 1/16" longer than it needs to be as the robot stands straight up; this ensures that the out step of the feet will be firm on the ground and aids with turning.

When you've got the length about right, snap the rod onto the foot and servo control horn (Figure 1-17)



**Figure 1-17**  
Install tilt rods.



Repeat for the other side.

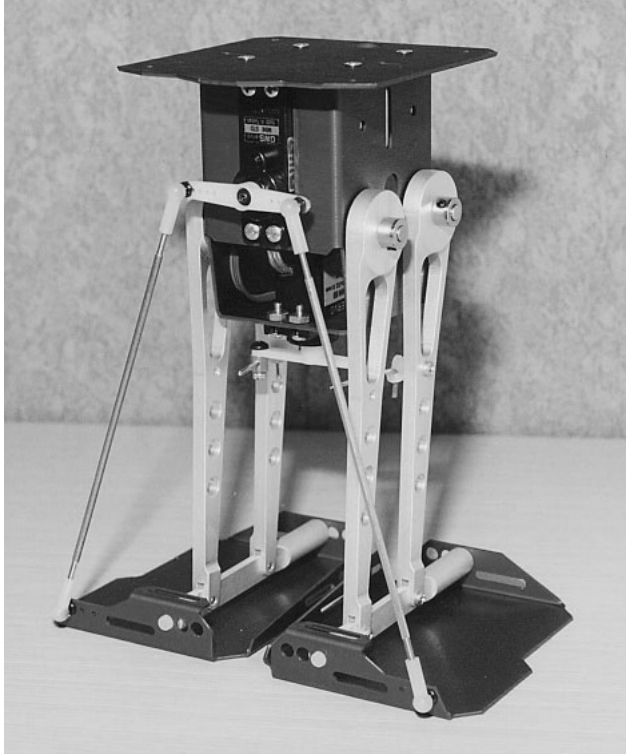
There is an easy way to remove the ball cups from the ball joint. If you need to make adjustments simply place a screwdriver between the ball cup and the Toddler's foot and carefully pry (snap) it off. It should pop off and can be pressed back on after you make a few turns to adjust.



**Toddler Tuning Tips:** Toddler functions best when its body is shifted slightly toward the rear feet. Also, with both servos centered verify that the Toddler's feet are flat on the ground or that the outsteps are angled slightly downward.

When you're done with this step your robot should look like the one shown in Figure 1-18. The Toddler functions best when its body is shifted slightly toward the rear feet.

When you pick up the robot, verify that the robot's feet are flat on the ground, or that the outsteps are angled slightly downward. Electronically center the servos with the BASIC Stamp again if needed to tune the hardware again.



**Figure 1-18**  
Final installation of tilt rods

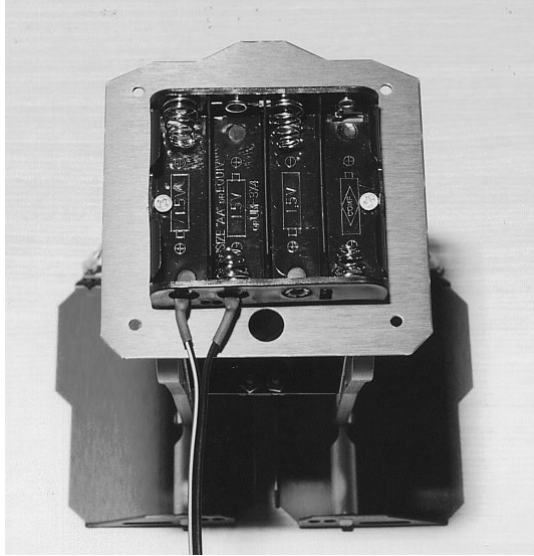
### **Step #15: Install Battery Pack**

The following parts are required for this step:

- Battery pack
- (2) 4/40 3/8" long flathead countersunk machine screws
- (2) 4/40 nuts

Stand the Toddler up on its feet.

Install the plastic battery pack using two 4/40 3/8" flathead screws and nuts (Figure 1-19). The flathead screws will be countersunk into the battery pack when tightened. The screws should be out of the way of the batteries.



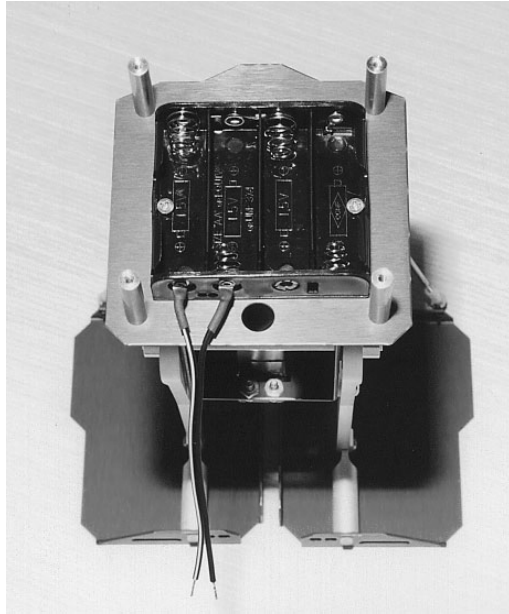
**Figure 1-19**  
Install Battery Pack.

### **Step #16: Install Standoffs**

The following parts are required for this step:

- (4) 1" female/female 4/40 standoffs
- (4) 4/40 ¼" panhead machine screws

Using four 4/40 ¼" screws install the four 1" standoffs on the top plate (Figure 1-20).



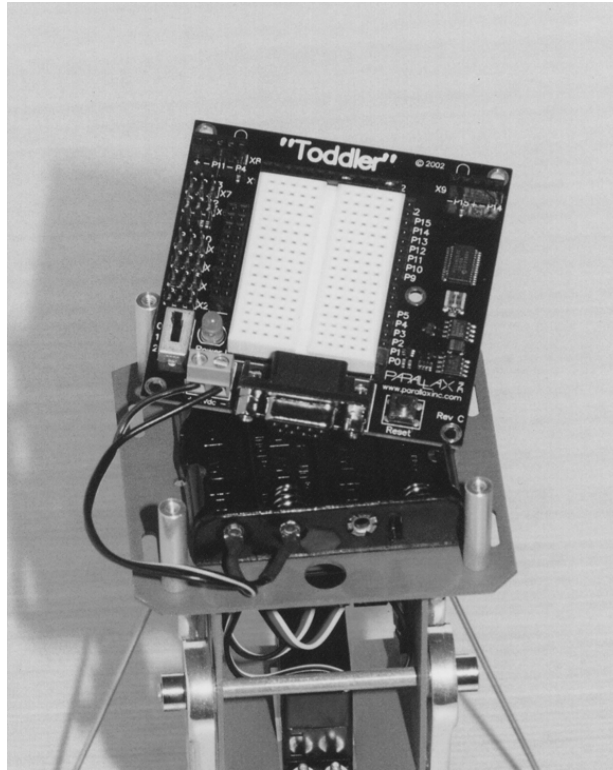
**Figure 1-20**  
Install standoffs on top plate.

**Step #17: Connect Toddler Board to Battery Pack**

The following parts are required for this step:

- Toddler BASIC Stamp printed circuit board

The battery pack's white lead connects to the Toddler board's + terminal. The other lead connects to the – terminal. Using a screwdriver secure both wires (Figure 1-21).



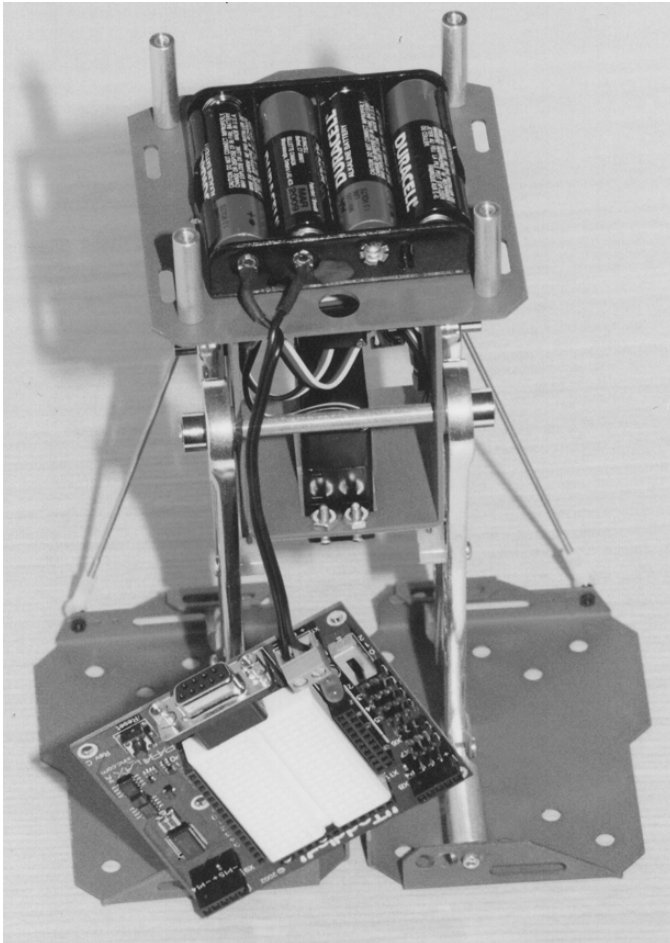
**Figure 1-21**  
Connect Toddler board to battery pack.

**Step #18: Install 4 AA Batteries**

The following parts are required for this step:

- (4) AA batteries (not included in your kit)

Install 4 AA batteries (Figure 1-22). Slide the Toddler's switch to Position 1 to verify that power is properly connected. The green power light will turn on.



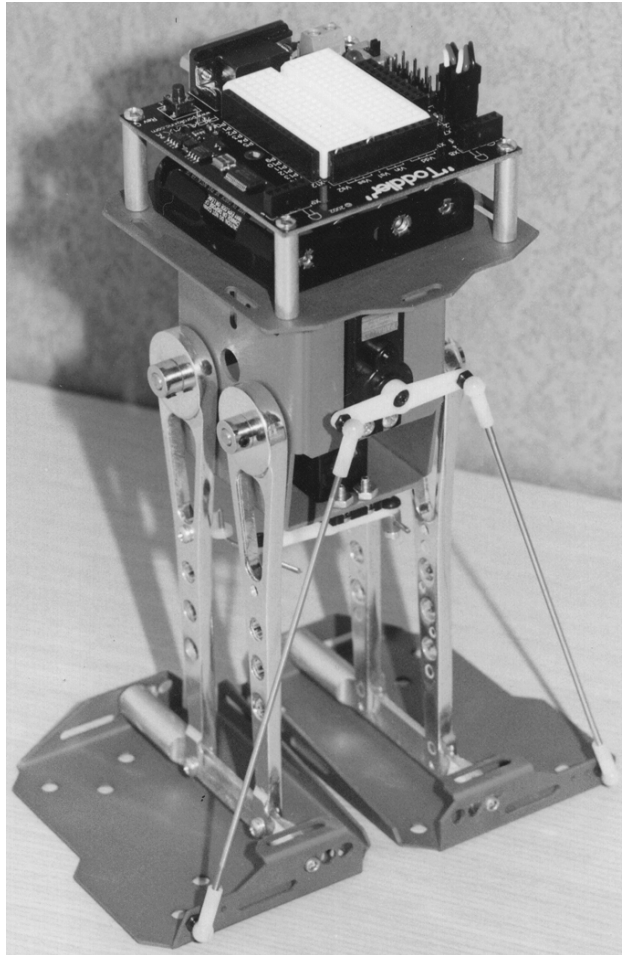
**Figure 1-22**  
Install 4AA batteries.

**Step #19: Mount Toddler Board**

The following parts are required for this step:

- (4) 4/40 1/4" panhead machine screws

Using four 4/40 1/4" panhead machine screws mount the Toddler board on the top of the standoffs (Figure 1-23). Connect the bottom servo (stride) to P12 on the Toddler board. Connect the front servo (tilt) to P13 on the Toddler board.



**Figure 1-23**  
Completed Toddler

### **Step #20: Center Servos Again and Make Adjustments**

As a final step, repeat Step #3 to center the servos. The Toddler should stand flat on the ground with both feet aligned. When you pick the robot up, the feet outsteps should be slightly tilted downward. Adjustments can be made to the ball joints. The easiest way to remove the ball joint from the socket is to carefully pry it away from the foot with the screwdriver. Toddlers also walk best when their legs are not vertical – tilt them back one or two degrees so the center of gravity is closer to the middle of the feet.



## Chapter #2: Taking Your First Steps

---

# 2

### SIMPLE CONTROL AND YOUR FIRST PROGRAM

Making the Toddler walk requires some patience – the Toddler has 36 different individual movements. In this experiment you'll learn how to make the robot walk forward and backward by writing several routines.

After forward and backward movements are mastered we'll try making some turns in Chapter #3. You'll see that linking movements requires attention to the previous step your Toddler took. For example, you can only move your left leg forward if it is off the ground.

When the basics are mastered, you'll learn to store movements and sub-movements in EEPROM and write more efficient code. All of this section is "open-loop" – that means there's no feedback to determine whether or not you have instructed your Toddler to lean too far left or right or even to look for obstacles.

### Servo Control Basics

#### How a Servo Works

Hobby servos are very popular for controlling the steering and throttle systems in radio-controlled cars, boats, and planes. These servos are designed to control the position of something such as a steering flap on a radio-controlled airplane. Their range of motion is typically  $90^\circ$  to  $270^\circ$ , and they are great for applications where inexpensive, accurate high-torque positioning motion is required. The position of these servos is controlled by an electronic signal called a pulse train, which you'll get some first hand experience with shortly. Hobby servos have built-in mechanical stoppers to prevent them from turning beyond the  $90^\circ$  or  $270^\circ$  range of motion. A hobby servo also has internal mechanical linkages for position feedback so that the electronic circuit that controls the DC motor inside the servo knows where to turn to in response to a pulse train. The Toddler uses unmodified servos which rotate through a range of motion about equal to  $270^\circ$ .

### Time Measurements and Voltage Levels

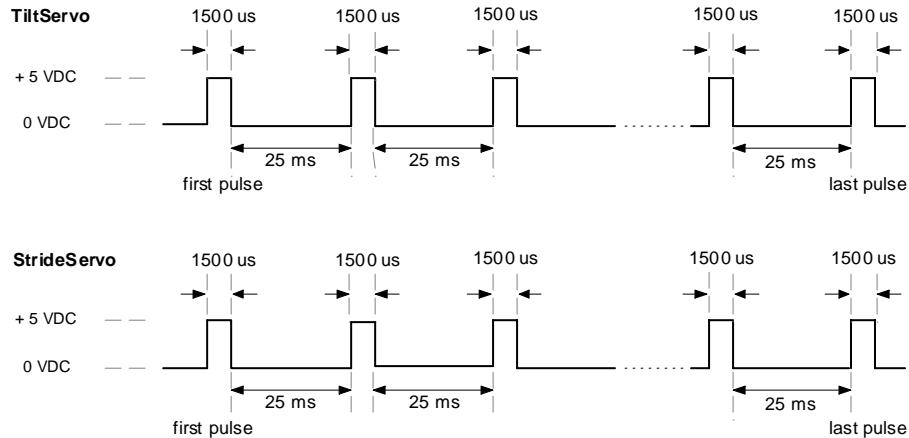
Throughout this student guide, amounts of time will be referred to in units of seconds (s), milliseconds (ms), and microseconds (us). Seconds are abbreviated with the lower-case letter “s”. So, one second is written as 1 s. Milliseconds are abbreviated as ms, and it means one one-thousandth of a second. One microsecond is one one-millionth of a second. Table 2-1 shows how Milliseconds and Microseconds equate in terms of both fractions and scientific notation.

<b>Table 2-1: Milliseconds and Microseconds</b>
$1 \text{ ms} = \frac{1}{1000} \text{ s} = 1 \times 10^{-3} \text{ s}$
$1 \text{ } \mu\text{s} = \frac{1}{1,000,000} \text{ s} = 1 \times 10^{-6} \text{ s}$

A voltage level is measured in volts, which is abbreviated with an upper case V. The Toddler board has sockets labeled Vss, Vdd, and Vin. Vss is called the system ground or reference voltage. When the battery pack is plugged in, Vss is connected to its negative terminal. Vin is unregulated 6 V (from four AA batteries), and it is connected to the positive terminal of the battery pack. Vdd is regulated to 5 V by the Toddler’s onboard voltage regulator, and it will be used with Vss to supply power to circuits built on the Toddler’s breadboard (Table 2-2).

<b>Table 2-2: Voltages and Toddler PCB Labels</b>
Vss = 0 V (ground) Vdd = 5V (regulated) Vin = 6 V (unregulated)

The control signal the BASIC Stamp sends to the servo's control line is called a "pulse train," and an example of one is shown in Figure 2-1. The BASIC Stamp can be programmed to produce this waveform using any of its I/O pins. In this example, the BASIC Stamp sends a 1500 us pulse to P12 (stride servo) and P13 (tilt servo). When the pulse is done being executed the signal pin is low. Then, the BASIC Stamp sends a 25 ms pause.



**Figure 2-1:** Example Servo Pulse Train Centers Servo

This pulse train has a 1500 us high time and a 25 ms low time. The high time is the main ingredient for controlling a servo's motion, and it is most commonly referred to as the pulse width. Since these pulses go from low to high (0 V to 5 V) for a certain amount of time, they are called positive pulses. Negative pulses would involve a resting state that's high with pulses that drop low.

The ideal pause between servo pulses can be about 10-40 ms without adversely affecting the servo's performance.



**The PULSOUT command** The BASIC Stamp 2's PULSOUT command works in increments of 2 microseconds. For example, the following snippet of code makes a 1500 us pulse:

```
PULSOUT 13, 750           ` 1.5 ms pulse on P13
```

### **Many Ways to Move the Toddler**

Programming is a cross between an art and science. There are usually many different ways a program can be written to get the same effect. Some are more efficient in program size and others are more efficient in performance.

In this chapter, we look at a number of different actions the Toddler can perform including walking forward and backward. The Toddler robot can make 36 different movements. These different movements must be linked together in order to walk. Each movement has a selection of possible precedent movements and a selection of possible follow-up movements.

We will also take a look at a number of different programs that perform these functions using different approaches. We present three approaches for programming the Toddler's movements. Most will prefer the last method. It uses more complicated programming techniques but it is more flexible and easier to use. Experienced programmers will want to jump right to the last approach but it is worth checking out all three.

#### **Approach #1: Brute Force**

This approach uses explicit subroutines for each movement. Calling these routines performs complex movements. It provides an obvious way of controlling the Toddler but enumerating all 36 movements consumes lots of precious program space. It also makes changes unwieldy. For example, implementing variable speed movements requires changes to all movement routines.

#### **Approach #2: Data Tables**

One obvious approach to consolidating 36 similar routines is to determine commonality within the routines and generating one or more routines that perform the same functions within certain parameters. Putting the parameters into data tables is one way to do this. Tables tend to be more concise in terms of construction compared to more explicit routines because the tables only contain parameters.

### **Approach #3: State Transitions**

The DATA tables approach is really a consolidation of the first and second approach. The programmer must remember where the robot's feet are and call the appropriate routine or fill in the table with the proper parameters. The state transition approach is different because the Toddler keeps track of where its feet are. Transition actions are now used to move from one state to another. There are basically three tilt and three stride actions. That is significantly less than the 36 movements used in the other approaches.

#### **Theory of Operation**

Humans take to walking naturally but the actual act is extremely complex. It requires the coordinated actions of muscles and these actions are controlled by a very complex brain with a very sophisticated array of inputs from vision to touch. The Toddler is at the other end of the spectrum. It has only two control servos with a limited range of motion and essentially no feedback. Although the Toddler will not learn to walk of its own accord, it can be programmed relatively easily once you understand the basics.

Humans usually walk using a controlled fall. The body tilts slightly forward and a leg is moved in front to stop the fall. The process is repeated as the person proceeds to move forward. The effect is more noticeable when a person is running. It is easy to see why a person falls on the ground if they misstep.

It is possible for humans to shuffle along like the Toddler. In this case the foot is placed in front and the body is pulled along but it is hard to do. Try it. Tilting your body forward makes it easier but this is essentially a controlled fall.

The Toddler can fall over but its movement is done via shuffling and balance. This is necessary because of the limited range of movement. Essentially the Toddler can lean to either side or stand flat with both feet on the floor. Either the left or right foot can be in front, in which case the other is in the back, or they can be side-by-side. There are essentially 9 basic foot orientations and there are 4 transitions from each orientation to another valid orientation for a total of 36 transitions or movements.

Even with this limited range of actions, the Toddler can move about a flat surface with relative ease. Its restrictions do limit the Toddler to two basic kinds of movements though: walking in a straight line and pivoting. Still, this can get the Toddler from point A to point B.

Walking is essentially a four phase process.

- a. Tilt to one side
- b. Move the leg that is not on the ground
- c. Tilt to the other side
- d. Move the leg that is not on the ground

This process essentially takes one step. The direction the leg moves controls the direction, and the distance traveled is controlled by how far the legs are moved. The speed of the Toddler depends on how fast the actions are performed and how far the legs move.

Assuming the Toddler is not programmed to tilt too far in one direction, it will remain balanced. This means the process can be stopped and restarted later at any point. This differs from humans in a controlled fall because the foot must be there to stop the fall.

Momentum plays a key part in writing programs to control the Toddler. The servo motors provide precise leg position control. They can move the legs slowly or quickly and can stop them at any location along the way. The Toddler also remains balanced even when tilted far to one side but this limit is difficult to attain when moving quickly because of momentum. If you speed up the leg movement, the amount of momentum the leg has increases. Trying to stop it at the balance limit is only possible if the leg has little momentum. Too much momentum at that point and the Toddler falls over.

For turns the Toddler can only pivot. It does not have a knee or hip joint like a human. The Toddler's feet always face forward so it cannot turn its feet to change direction. This does not restrict the Toddler to straight line motion though. By pivoting, the Toddler can move from Point A to Point B in a straight line, pivot in the direction of Point C and then walk in that direction to Point C.

Pivoting is also a four-phase process.

- a. Tilt to one side
- b. Move the leg that is not on the ground
- c. Put both feet on the ground
- d. Move the legs opposite of each other

This process works because of friction. The actual pivot occurs in the last phase where both feet are on the ground. Essentially one leg pulls the Toddler forward while the other pulls it backward. This causes the Toddler to pivot. The amount of rotation is a factor of leg placement and the level of friction between the Toddler's feet and the surface it is on. Low friction results in minimal pivoting. Too much friction and the Toddler can fall over.

The Toddler is essentially restricted to flat surfaces. The type of surface affects the amount of friction between the Toddler and the surface. Wood, hard carpet and kitchen linoleum floors are a good surface while ice and rubber are not. Dense carpet works well. Putting different surfaces on the bottom of the Toddler's feet can increase the level of friction. This is typically done using tape (electrical, maybe even a small piece of grip tape). There are not hard rules on choosing surfaces and increasing friction so experiment. You may have to adjust the program controlling the Toddler to take into account the surface. For example, you cannot assume that two pivot actions will turn the Toddler 90 degrees.

This brings up the issue of accuracy. The Toddler does a good job of moving but it is relatively inaccurate in its movements compared to its wheeled cousin, the Boe-Bot. If the Toddler goes six steps forward and six steps back it will not wind up in exactly the same spot. It may be close but it is unlikely to be exact. Add some turns and all bets are off. Getting the Toddler to walk in a square is next to impossible. It is easy to program the commands to walk in a square but due to friction and the mechanical accuracy of the Toddler's movements, the Toddler will probably not walk over its own footsteps. Unless you've tried to add encoders to a wheeled robot or tried to solve a maze, the Toddler's relative limitations of repeatability are of little concern.

For most experiments, accuracy is not an issue. It is possible to track the Toddler's orientation using the optional Compass AppMod but tracking distance moved is a daunting task at best. These problems are beyond the scope of this book but great areas for investigation. Answering the question "where is my robot?" is one of the most challenging hobby and educational endeavors, whether it uses wheels or legs.

The lack of articulated legs prevents the Toddler from walking over obstacles. The Toddler cannot handle a grade of any significance so stay away from ramps. The Toddler can avoid obstacles by going around them. In later experiments, we examine obstacle detection using infrared devices included with the Toddler.

### ACTIVITY #1: BASIC WALKING MOVEMENTS: BRUTE FORCE

The Toddler is a walking robot so getting it moving is a good starting point. Figure 2-2 shows a possible order of operation for taking a few steps forward.

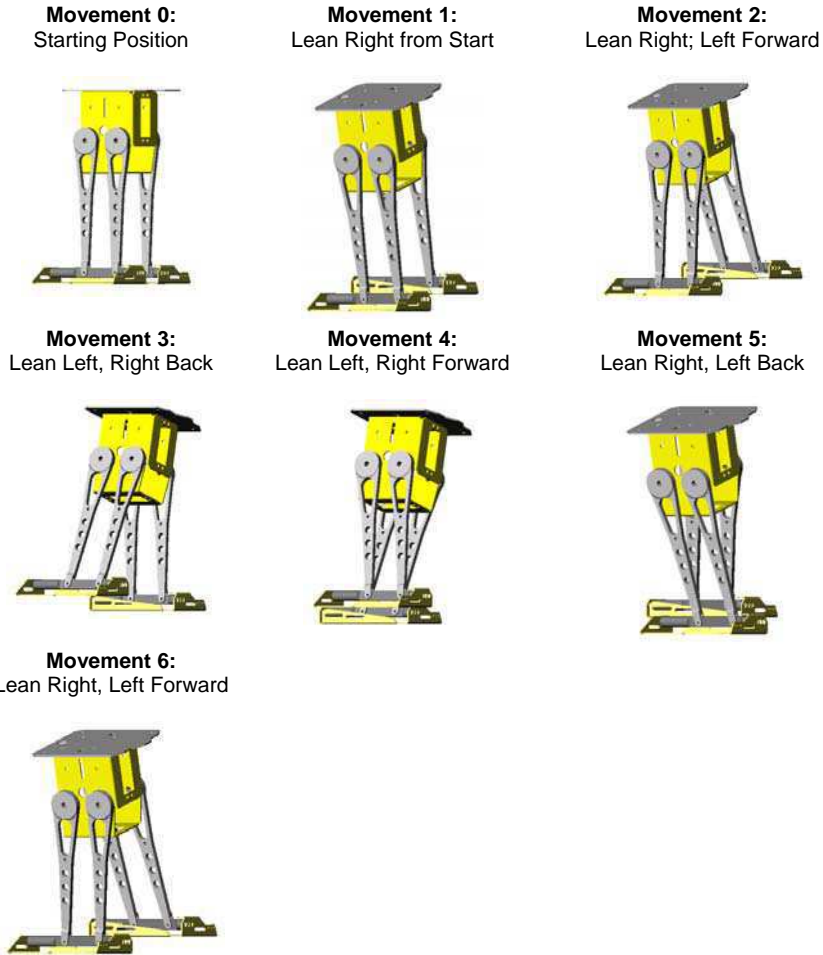


Figure 2-2: First Steps Movement Patterns



Once the Toddler has walked Movements 0, 1 and 2 the process of Movements 3, 4, 5 and 6 can repeat to walk in a straight line. The code to perform the first three movements is shown in the following pages. Movements 1 and 2 are almost identical except that one adjusts the tilt and the other the stride. Movements 1, 3 and 5 would use the same code with different values. The same is true for movements 2, 4 and 6.

The **m0** routine is unique. It is designed to place the feet of the Toddler on the ground and next to each other regardless of where feet are when the routine starts. This can result in a jerking motion if the legs are not already in or close to this point.

The **m1** and **m2** routines are representative examples of all the other movement routines in the program. The **m1** movement tilts the Toddler to the right. To do so, it sends pulses to both servos. It sends the same pulse width to the stride motor so it remains stationary and the feet do not move forward or backwards. The tilt servo is sent pulses that progressively change in width causing the tilt servo to rotate which in turn causes the Toddler to lean.

The **m2** routine does the same thing but in this case the tilt servo is held stationary while the stride servo is driven by a set of different width pulses causing one leg to move ahead of the other.

The program uses constant named definitions for the range of servo pulse width limit values while the examples with small snippets of code use numeric constant values. The effect is the same but the named constants minimize changes to the program when experimenting with different values.

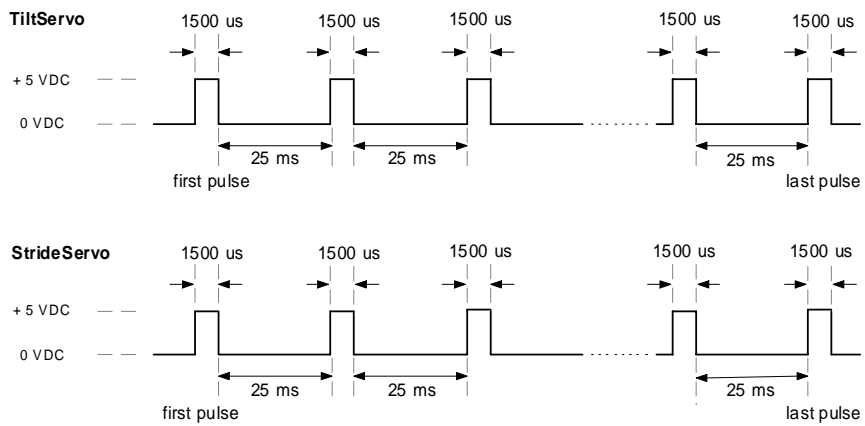
The **FOR...NEXT** loop uses a **STEP** modifier in the following pictorial examples. This **STEP** does not refer to the number of physical Toddler steps taken in the routine, but an increment to the pulse sent to the servo to send it through the desired range of motion.



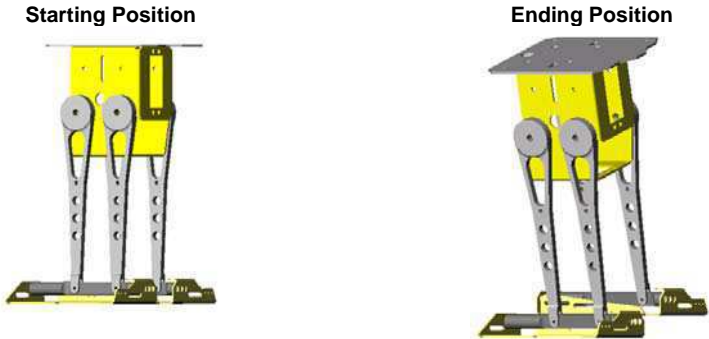
**Movement 0 BASIC Stamp Code**

```
M0:  
FOR Pulses = 1 TO 100 STEP 5  
  PULSOUT TiltServo, 750  
  PULSOUT StrideServo, 750  
  PAUSE 25  
NEXT
```

**Movement 0 Timing Diagram**



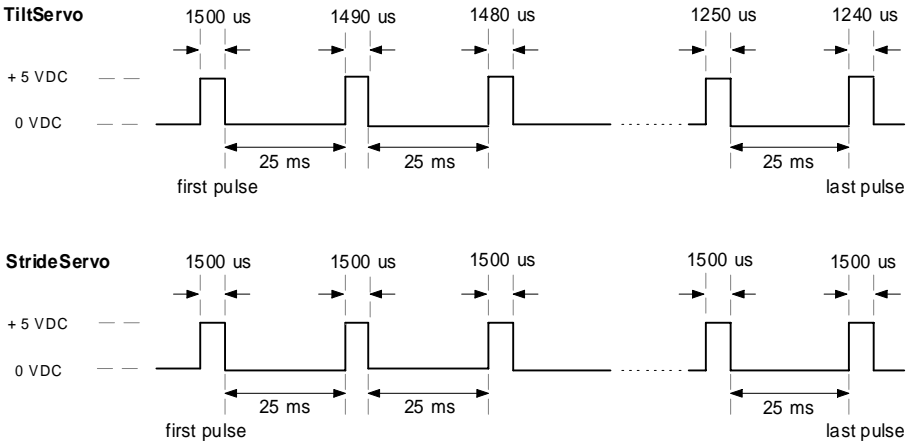
**Figure 2-3: Movement 0 Example**



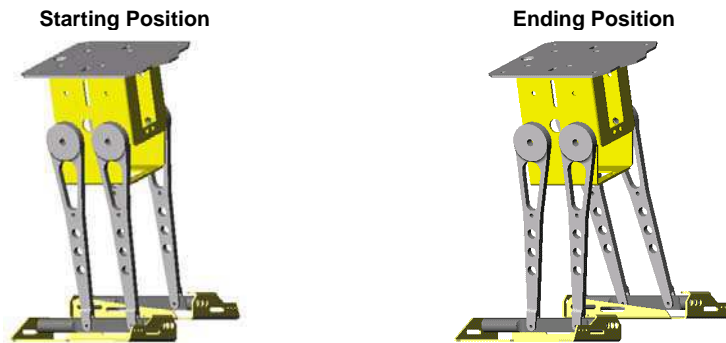
**Movement 1 BASIC Stamp Code**

```
M1:  
  FOR Pulses = 750 TO 620 STEP 5  
    PULSOUT TiltServo, Pulses  
    PULSOUT StrideServo, 750  
    PAUSE 25  
  NEXT
```

**Movement 1 Timing Diagram**



**Figure 2-4: Movement 1 Example**

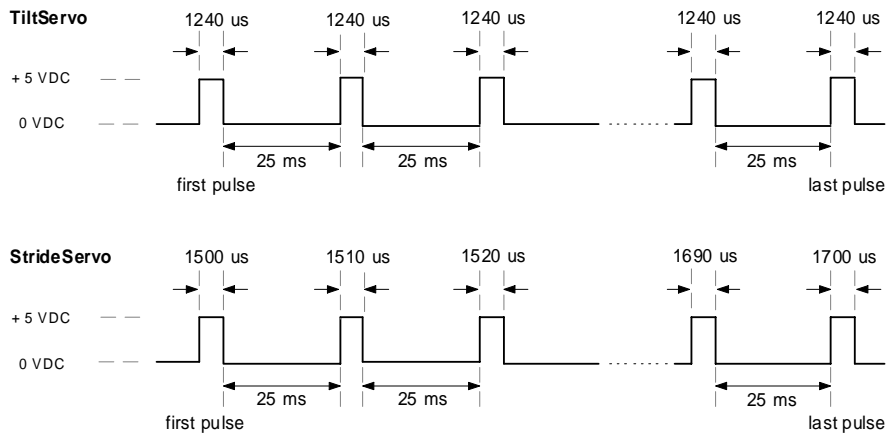


**Movement 2 BASIC Stamp Code**

```

M2:
  FOR Pulses = 750 TO 850 STEP 5
    PULSOUT TiltServo, 620
    PULSOUT StrideServo, Pulses
    PAUSE 25
  NEXT
  
```

**Movement 2 Timing Diagram**



**Figure 2-5: Movement 2 Example**

An example program that performs all the movements for walking forward is shown below. It is possible to adjust the different constants in the program to make the robot walk faster or take bigger steps. Be careful of large changes because the Toddler can fall over. More on that later.

This program also cleans up its movement so both feet are centered and flat on the floor when it is done. This makes starting other programs easier since the feet are in a known position.

```
' -----[ Title ]-----
' Toddler Program 2.1 - First Steps Forward.bs2
' Run Movement Patterns M0 to M8 to take several steps
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Declarations ]-----

TiltStep      CON    5           ' TiltServo step size
StrideStep    CON    5           ' StrideServo step size
MoveDelay     CON   25           ' in microseconds

RightTilt     CON   620          ' Tilt limits
CenterTilt    CON   750
LeftTilt      CON   880

RightForward  CON   650          ' Stride limits
StrideCenter  CON   750
LeftForward   CON   850

StrideServo   CON    12          ' Stride servo on P12
TiltServo     CON    13          ' Tilt servo on P13

MoveLoop      VAR    Nib         ' Repeat movements
Pulses        VAR    Word        ' Pulse variable

' -----[ Initialization ]-----

' -----[ Main Routine ]-----
' Take three full steps.

Main_Program:
  GOSUB M0           ' center servos
  GOSUB M1           ' tilt right
  GOSUB M2           ' step left
  FOR MoveLoop = 1 to 3
    GOSUB M3         ' tilt left
    GOSUB M4         ' step right
```

```

        GOSUB M5                ' tilt right
        GOSUB M6                ' step left
    NEXT
    GOSUB M3                    ' tilt left
    GOSUB M7                    ' center feet
    GOSUB M8                    ' center servos
END

' -----[ Subroutines ]-----
M0:
    FOR Pulses = 1 TO 100 STEP StrideStep
        PULSOUT TiltServo, CenterTilt
        PULSOUT StrideServo, StrideCenter
        PAUSE MoveDelay
    NEXT
    RETURN

M1:
    FOR Pulses = CenterTilt TO RightTilt STEP TiltStep
        PULSOUT TiltServo, Pulses
        PULSOUT StrideServo, StrideCenter
        PAUSE MoveDelay
    NEXT
    RETURN

M2:
    FOR Pulses = StrideCenter TO LeftForward STEP StrideStep
        PULSOUT TiltServo, RightTilt
        PULSOUT StrideServo, Pulses
        PAUSE MoveDelay
    NEXT
    RETURN

M3:
    FOR Pulses = RightTilt TO LeftTilt STEP TiltStep
        PULSOUT TiltServo, Pulses
        PULSOUT StrideServo, LeftForward
        PAUSE MoveDelay
    NEXT
    RETURN

M4:
    FOR Pulses = LeftForward TO RightForward STEP StrideStep
        PULSOUT TiltServo, LeftTilt
        PULSOUT StrideServo, Pulses
        PAUSE MoveDelay
    NEXT
    RETURN

M5:

```

```
FOR Pulses = LeftTilt TO RightTilt STEP TiltStep
  PULSOUT TiltServo,Pulses
  PULSOUT StrideServo, RightForward
  PAUSE MoveDelay
NEXT
RETURN

M6:
FOR Pulses = RightForward TO LeftForward STEP StrideStep
  PULSOUT TiltServo,RightTilt
  PULSOUT StrideServo, Pulses
  PAUSE MoveDelay
NEXT
RETURN

M7:
FOR Pulses = LeftForward TO StrideCenter STEP StrideStep
  PULSOUT TiltServo,LeftTilt
  PULSOUT StrideServo, Pulses
  PAUSE MoveDelay
NEXT
RETURN

M8:
FOR Pulses = LeftTilt TO CenterTilt STEP TiltStep
  PULSOUT TiltServo,Pulses
  PULSOUT StrideServo, StrideCenter
  PAUSE MoveDelay
NEXT
RETURN
```

Note that the program is downloaded to the Toddler using the serial cable with the power switch in either the download (1) or run (2) position. The cable can remain connected to the Toddler while it is walking if there is sufficient length to allow it to move freely about the PC. You may want to hold the cable near the Toddler as it does make it slightly less stable. The power on the Toddler should be turned off when disconnecting the cable. The Toddler can then walk on its own when the power is turned on since the program is downloaded into non-volatile memory. You can also program the Toddler with the switch in Position 1, hold the reset button and move it to Position 2 so it will walk.

Three constants could be modified to make it walk quicker: **MoveDelay** and **TiltStep** and **StrideStep**. Decreasing **MoveDelay** means there will be less pause between the servo pulses. Increasing **TiltStep** and **StrideStep** means the servo pulse width changes will be larger (making for fewer pulses to complete the step).

If the Toddler isn't starting with both feet firmly planted squarely on the ground, or if you would like to experiment with larger step distances you could modify the `CenterTilt` and `StrideCenter` values. This would result in a need to also modify the right and left limits for both the tilt and stride.

### **Your Turn**

Modify the code in the example. Try the following:

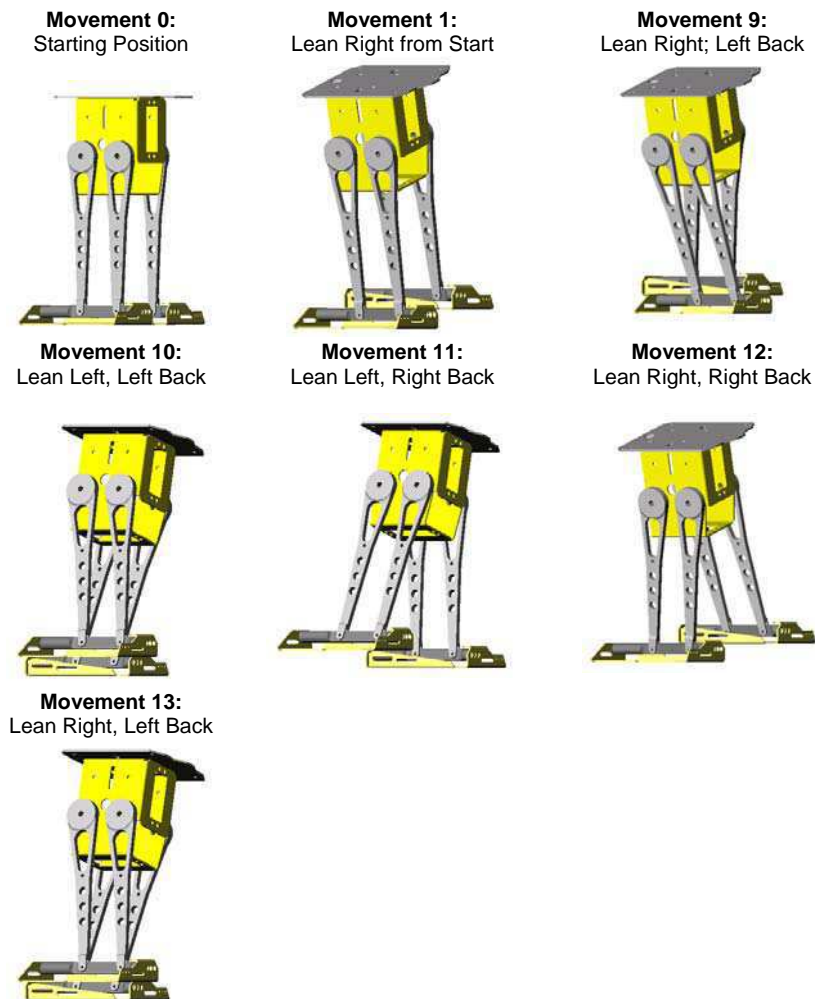
- √ Adjust the `TiltStep` and `StrideStep` values to smaller numbers. Observe how the Toddler behaves. Adjust the same constants to larger numbers and make some observations. Can you envision the servo timing diagram and explain why the Toddler behaves differently?
- √ Decrease the `MoveDelay` value to a smaller number and make observations. Find the right combination of `MoveDelay`, `TiltStep` and `StrideStep` that makes your Toddler walk the way you want it to. Save these values for future programs.



**ACTIVITY #2: WALKING BACKWARDS: BRUTE FORCE**

The Toddler robot can walk backward as well as forward but it is not simply a matter of using the steps in the prior program in reverse order. The Toddler moves in the reverse fashion but the functions necessary to do this will be different. With just over half a dozen routines, the last sample program is relatively simple. Changing it to handling a different direction is not too difficult. Keep in mind that the two other approaches to performing these tasks are presented later in this chapter.

In the prior program, the subroutine for each step was numbered sequentially. In this program, the steps will be slightly different so we can use different routine names. The starting movement is the same as the prior program but the second step will be Movement 9 that matches the `m9` routine.



**Figure 2-6:** First Steps – in Reverse

As with the forward walking program, the Toddler starts with Movements 0, 1 and 9. The process of Movements 10, 11, 12 and 13 can repeat to walk in a straight line but

backwards. An example program is shown below. Adjust the different constants in the program to make your robot walk faster or take bigger steps. The program also cleans up its movement so both feet are centered and flat on the floor.

Note that the routines with the same name have been extracted from the first sample program. A program that used this approach but required more sophisticated actions would need more routines with the potential of requiring all 36.

```
' -----[ Title ]-----
' Toddler Program 2.2 - Walking Backwards.bs2
' Run Movement Patterns M9 to M13 to walk backwards
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Declarations ]-----

TiltStep      CON    5          ' TiltServo step size
StrideStep    CON    5          ' StrideServo step size
MoveDelay     CON   20          ' in microseconds

RightTilt     CON   620         ' Tilt limits
CenterTilt    CON   750
LeftTilt      CON   880

RightForward  CON   650         ' Stride limits
StrideCenter  CON   750
LeftForward   CON   850

StrideServo   CON    12        ' Stride servo on P12
TiltServo     CON    13        ' Tilt servo on P13

MoveLoop      VAR    Nib        ' Repeat movements
Pulses        VAR    Word       ' Pulse variable

' -----[ Initialization ]-----

' -----[ Main Routine ]-----

Main_Program:
  GOSUB M0          ' center servos
  GOSUB M1          ' tilt right
  GOSUB M9          ' step back

  FOR MoveLoop = 1 to 3
    GOSUB M10       ' tilt left
    GOSUB M11       ' step left
    GOSUB M12       ' tilt right
    GOSUB M13       ' step right
```

```

NEXT

GOSUB M10           ' tilt left
GOSUB M14           ' center feet
GOSUB M8            ' center servos
END

' -----[ Subroutines ]-----
M0:
  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, StrideCenter
    PAUSE MoveDelay
  NEXT
  RETURN

M1:
  FOR Pulses = CenterTilt TO RightTilt STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, StrideCenter
    PAUSE MoveDelay
  NEXT
  RETURN

M8:
  FOR Pulses = LeftTilt TO CenterTilt STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, StrideCenter
    PAUSE MoveDelay
  NEXT
  RETURN

M9:
  FOR Pulses = StrideCenter TO RightForward STEP StrideStep
    PULSOUT TiltServo, RightTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT
  RETURN

M10:
  FOR Pulses = RightTilt TO LeftTilt STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, RightForward
    PAUSE MoveDelay
  NEXT
  RETURN

M11:
  FOR Pulses = RightForward TO LeftForward STEP StrideStep

```

```

    PULSOUT TiltServo,LeftTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT
  RETURN

M12:
  FOR Pulses = LeftTilt TO RightTilt STEP TiltStep
    PULSOUT TiltServo,Pulses
    PULSOUT StrideServo, LeftForward
    PAUSE MoveDelay
  NEXT
  RETURN

M13:
  FOR Pulses = LeftForward TO RightForward STEP StrideStep
    PULSOUT TiltServo,RightTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT
  RETURN

M14:
  FOR Pulses = RightForward TO StrideCenter STEP StrideStep
    PULSOUT TiltServo,LeftTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT
  RETURN

```

### **Your Turn**

Try the following:


- √ Experiment with the constants in the program by changing them to the following values. Explain why the Toddler's behavior is different.
 

TiltStep	CON	2	'	TiltServo step size
StrideStep	CON	2	'	StrideServo step size
MoveDelay	CON	12	'	in microseconds
- √ Write a program where the Toddler walks forward, then backward to the starting position.

### ACTIVITY #3: USING A DATA TABLE TO STORE MOVEMENTS

The length of the two prior programs is similar but more complex programs could grow larger simply because more movement routines would be necessary. Moving information in tables is an excellent way of simplifying the programming task. This next sample application employs both fixed size and variable length tables.

Table data is stored using the BASIC Stamp's **DATA** command. This command lets you write data to the Stamp's EEPROM during program download. If used correctly, it can make your PBASIC programs much more efficient and shorter, too.



**The DATA command** is shown with many examples in the BASIC Stamp Programming Manual, available on [www.parallax.com](http://www.parallax.com) for download or in a printed format. The basics are the following:

- **DATA** can be written in byte (8-bit) and word (16-bit) sizes. The BASIC Stamp 2's EEPROM stores DATA from numeric location 0 and builds towards location 2048. Your PBASIC program is stored from 2048 and writes backwards towards location 0.
- The PBASIC editor does not detect occurrences when you overwrite your own code with **DATA**.
- Once you download and run Toddler Program 2.3 – First Steps Forward Using Tables.bs2 you can choose Run | Memory Map to see how **DATA** and program is stored in the EEPROM.

This example uses two types of entries to develop walking movements. The following table entries are a list of movements that comprise the given motion. For example, we know from the prior two programs that **m1** followed by an **m2** movement will start the Toddler off walking. Then, after this is done movements 3, 4, 5, and 6 can repeat themselves so the Toddler can walk straight.

```

StartForward  DATA  1, 2, 0
WalkForward  DATA  3, 4, 5, 6, 0
FinishForward DATA  3, 7, 8, 0
    
```

The BASIC Stamp's EEPROM stores the byte-sized value 1 in location 0, the value 2 in location 1, the value 0 in location 3, the value 3 in location 4 and so on.

The variable length tables are used to store a sequence of movements. This allows complex movements to be of arbitrary length. The tables are accessed using the PBASIC **READ** command.

The **Movement** contains data that is used to tilt or stride the Toddler. Each movement has a starting and ending tilt and stride value. In this **DATA** table, **M1** is actually an address for the storage of **CenterTilt**. Because the above three lines of code had 12 byte-sized entries, the **CenterTilt** word-size constant will be stored across EEPROM locations 12 and 13 (the starting EEPROM location is 0, not 1).

The **WORD** modifier is used because the constants are all greater than 255 – the value of a byte.

M1	DATA	WORD CenterTilt, WORD RightTilt, WORD StrideCenter, WORD StrideCenter
M2	DATA	WORD RightTilt, WORD RightTilt, WORD StrideCenter, WORD LeftForward
M3	DATA	WORD RightTilt, WORD LeftTilt, WORD LeftForward, WORD LeftForward
M4	DATA	WORD LeftTilt, WORD LeftTilt, WORD LeftForward, WORD RightForward
M5	DATA	WORD LeftTilt, WORD RightTilt, WORD RightForward, WORD RightForward
M6	DATA	WORD RightTilt, WORD RightTilt, WORD RightForward, WORD LeftForward
M7	DATA	WORD LeftTilt, WORD LeftTilt, WORD LeftForward, WORD StrideCenter
M8	DATA	WORD LeftTilt, WORD CenterTilt, WORD StrideCenter, WORD StrideCenter

Only eight (8) movements are employed in this program but it is easy to see how all 36 movements could be easily added to the table. Storing the movements in EEPROM using **DATA** requires much less code space than the “brute force” first example.

The program’s management of these routines is handled in the **Main\_Program:** and **Movement:** routines. The **Main\_Program:** routine starts by calling the **M0** routine, which centers the Toddler’s stride and tilt so both feet are place squarely on the surface.

On return, the **Mx** variable is loaded with the value of **StartForward**, which is actually address 0 given by the **DATA** statement. In the **Movement:** routine the value of EEPROM address is **READ** and stored in **Dx**. **Dx** now equals 1. **Mx** is incremented by one for the next loop.

The first time through the program the `LOOKUP` table looks up index 1, storing the address of `M1` in the variable `Dx`. Then, four word values are read from the `Dx` address and stored consecutively in the `TiltStart`, `TiltEnd`, `StrideStart` and `StrideEnd` variables. These values are now used in the `PULSOUT` command to develop a movement.

The line of code:

```
IF TiltStart = TiltEnd THEN MovementStride
```

detects an instance when Toddler does not change the tilt, but only moves a leg forward. In this case, the previous `TiltStart` value is used to hold the tilting servo steady and only a stride occurs.

```
' -----[ Title ]-----
' Toddler Program 2.3: First Steps Forward Using Tables
' Movement routines stored in EEPROM using DATA statement
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
' Connect Vs1 and Vs2 to Vss to ground both servos

StrideServo    CON    12                ' Stride servo
TiltServo      CON    13                ' Tilt servo

' -----[ Constants ]-----

TiltStep       CON    10                ' TiltServo step size
StrideStep     CON    10                ' StrideServo step size
MoveDelay      CON    14                ' Servo pause (ms)

RightTilt      CON    625               ' Tilt limits
CenterTilt     CON    750
LeftTilt       CON    875

RightForward   CON    600               ' Stride limits
StrideCenter   CON    750
LeftForward    CON    900

' -----[ Variables ]-----

MoveLoop       VAR    Nib                ' Loop for repeat movements
Pulses         VAR    Word               ' Pulse variable

Dx             VAR    Pulses              ' Stores Mx movement index
Mx             VAR    Word               ' Movement index
TiltStart      VAR    Word               ' Start tilt value
```



```

TiltEnd      VAR      Word      ' End tilt value
StrideStart  VAR      Word      ' Start stride value
StrideEnd    VAR      Word      ' End stride value

' -----[ EEPROM Data ]-----
' Indicates which movement routines are executed to comprise steps
' Extend WalkForward with 3, 4, 5, 6, for longer walks
' The number 0 marks the end of a list

StartForward DATA 1, 2, 0
WalkForward  DATA 3, 4, 5, 6,3, 4, 5, 6,3, 4, 5, 6,3, 4, 5, 6,3, 4, 5,
6, 0
FinishForward DATA 3, 7, 8, 0

' Movement routines

M1          DATA  WORD CenterTilt, WORD RightTilt,
                  WORD StrideCenter, WORD StrideCenter
M2          DATA  WORD RightTilt, WORD RightTilt,
                  WORD StrideCenter, WORD LeftForward
M3          DATA  WORD RightTilt, WORD LeftTilt,
                  WORD LeftForward, WORD LeftForward
M4          DATA  WORD LeftTilt, WORD LeftTilt,
                  WORD LeftForward, WORD RightForward
M5          DATA  WORD LeftTilt, WORD RightTilt,
                  WORD RightForward, WORD RightForward
M6          DATA  WORD RightTilt, WORD RightTilt,
                  WORD RightForward, WORD LeftForward
M7          DATA  WORD LeftTilt, WORD LeftTilt,
                  WORD LeftForward, WORD StrideCenter
M8          DATA  WORD LeftTilt, WORD CenterTilt,
                  WORD StrideCenter, WORD StrideCenter

' -----[ Main Routine ]-----

Main_Program:
  GOSUB M0                                ' Center servos

  Mx = StartForward
  'DEBUG ? Mx
  GOSUB Movement
  FOR MoveLoop = 1 to 3
    Mx = WalkForward
    GOSUB Movement
  NEXT
  Mx = FinishForward
  GOSUB Movement
END

' -----[ Subroutines ]-----

```

```

Movement:
  READ Mx, Dx                                     ' Read state table number
  Mx = Mx + 1
  'DEBUG "Movement routine = ", DEC Mx,cr
  IF Dx = 0 THEN DoReturn                         ' Skip if no more states
  LOOKUP Dx, [M1, M1, M2, M3, M4, M5, M6, M7, M8], Dx
  READ Dx, WORD TiltStart, WORD TiltEnd,
           WORD StrideStart, WORD StrideEnd' Read table entry; store
  IF TiltStart = TiltEnd THEN MovementStride
  FOR Pulses = TiltStart TO TiltEnd STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, StrideStart
    PAUSE MoveDelay
  NEXT
  GOTO Movement

MovementStride:
  FOR Pulses = StrideStart TO StrideEnd STEP StrideStep
    PULSOUT TiltServo, TiltStart
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT
  GOTO Movement

' ----- M0: Move feet to initial center position -----

M0:
  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, StrideCenter
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN

```

Adding support for walking backward is easier with this program. Three things need to be changed. First, the extra movement entries must be added to the fixed size table and the `LOOKUP` command. Second, a variable length table must be added for stepping through a backward foot-step. Finally, the table name and a call to the movement routine must be added. This is significantly better than adding more routines to the program.

### **Your Turn**

Try the following:

- √ Modify the `TiltStep`, `StrideStep` and `MoveDelay` constants so the Toddler walks as smooth as possible, but not necessarily quickly.
- √ Modify the `WalkForward` data so that the Toddler walks additional steps.

### ACTIVITY #4: USING STATE TRANSITIONS FOR MOVEMENTS

Now we take a look at a completely different way of handling movement control. The prior two approaches essentially required the programming to string together a series of subroutine calls in the proper order. In those cases, the programmer knew what the prior state was and would only use movement routines that were relevant. It was relatively simple for the basic actions examined thus far but it gets very tedious as the actions become more complex.

The state transition approach presented here changes the kinds of routines used from ones that have a known starting and ending state to routines that indicate a desired ending state. The program keeps track of the current state and adjusts accordingly. The reason that this approach simplifies things is that there are really only two kinds of actions, tilt and stride, that the Toddler can perform and only three logical orientations with each of these for a total of nine states. Figure 2-7 shows these states.

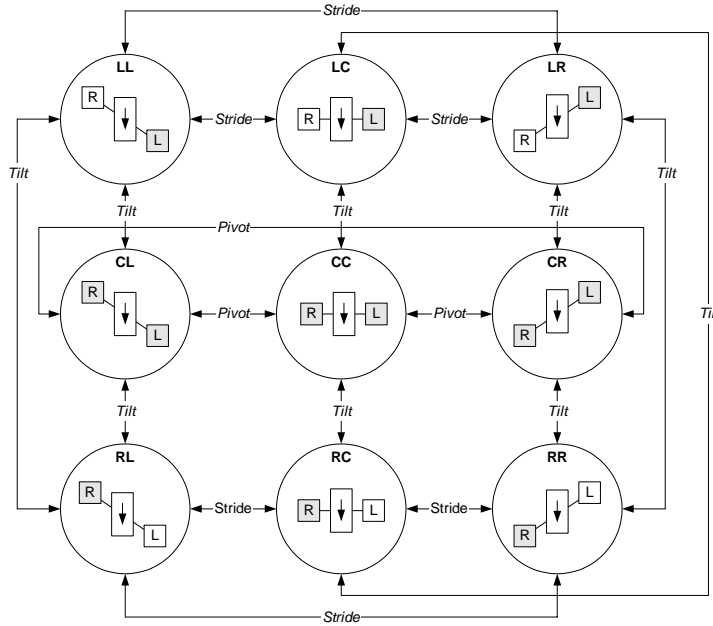


Figure 2-7: State Transition Approach

Each circle represents a state. The center shows the orientation of the Toddler's legs and body. The arrow indicates the front of the Toddler. The dark background in a rectangle indicates that a leg is on the floor. The white background indicates the leg is in the air. The two letters at the top of the circle provide a representation for the Toddler positioning. The first letter indicates the tilt (T) and the second indicates stride (S) or what foot is in front. The letters can be L, C, or R. C indicates the respect servo motor is centered. For the tilt, C indicates both feet are on the ground. For stride, C indicates that both feet are inline along the center of the body's axis. The reason for naming each state will become apparent soon.

There are also bi-directional arrows showing valid transitions from one state to another. The label on the line indicates the type of change that occurs. The Pivot designation is the same as Stride but we make the distinction because this will cause the Toddler to pivot.

Nine states, each with four possible transitions yields 36 distinct transitions. If the transition routines are labeled using their starting and ending state names with the format **TSxTS** then movement **M0** is **CCxRC** as it starts with both legs together in state **CC** and ends up leaning to the right in state **RC**. Using this naming convention makes programming easier in the prior examples with the code looking like this.

```
' -----[ Main Code ]-----
'
' Take three full steps.

Main_Program:
  GOSUB M0                ' center servos

  GOSUB CCxRC            ' tilt right
  GOSUB RCxRL            ' step left

  FOR MoveLoop = 1 to 3
    GOSUB RLxLL          ' tilt left
    GOSUB LLxLR          ' step right
    GOSUB LRxRR          ' tilt right
    GOSUB RRxRL          ' step left
  NEXT

  GOSUB RLxLL            ' tilt left
  GOSUB LLxLC            ' center feet
  GOSUB LCxCC            ' center servos
END
```

The advantage of the name change is obvious. The last two letters of the prior routine are the starting two letters of the next routine.

Of course, this is still relatively cryptic and without the comments the actions would definitely be confusing. The problem is that this approach requires knowledge of the prior state.

Now take a look at the next sample code snippet. It does away with the comments (not always a good idea but fine for this discussion) and the routines being called indicate what action is being performed.

```
' -----[ Main Code ]-----
'
' Take three full steps.

Main_Program:
  GOSUB M0           ' center servos

  GOSUB TiltRight
  GOSUB StrideLeft

  FOR MoveLoop = 1 to 3
    GOSUB TiltLeft
    GOSUB StrideRight
    GOSUB TiltRight
    GOSUB StrideLeft
  NEXT

  GOSUB TiltLeft
  GOSUB StrideCenter
  GOSUB TiltCenter
END
```

So what happened? You need to take a look at the next program listing to see how these routines were implemented but essentially two variables, **CurrentTilt** and **CurrentStride**, were added to keep track of where the feet are. The routine simply applies the designated change.

There are only six routines that need to be used. These can be applied in any order although only four will cause a change at any time.

Why? Because two of the six will cause the Toddler to stay in the state that it is currently in. For example, if the Toddler is tilting to the left then calling the **TiltLeft** routine will

leave it in the same position. There will be a delay while it executes the routine but it turns out to be a very short period of time. It is not noticeable when watching the Toddler move.

The following program implements the routines used in the prior code snippet but it retains some of the ideas employed in the last full program listing (Program 2.3) that used tables. They are still worthwhile although they are used in a slightly different fashion here. This program essentially connects the two prior complete program listings in that it lets the programmer connect a series of movements which are read from a **DATA** table yet it also reads the commands from an EEPROM. You will notice after you modify this program how the State Transition Approach figure applies.

To aid in your understanding of what the different **DATA** codes mean see Table 2-3.

<b>Table 2-3: State Table Codes</b>	
TR	tilt right
SL	stride left
TL	tilt left
SR	stride right
TC	tilt center
Xx	end of table

```

' -----[ Title ]-----
' Toddler Program 2.4: First Steps Forward with State Transitions
' Keeps track of starting and ending states
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      CON      13          ' Tilt servo on P12
StrideServo    CON      12          ' Stride servo on P13

' -----[ Constants ]-----

MoveDelay      CON      12          ' in microseconds
StrideStep     CON      5           ' StrideServo step size
TiltStep       CON      5           ' TiltServo step size

RightTilt      CON      620         ' Tilt limits
CenterTilt     CON      750
LeftTilt       CON      880

RightStride    CON      650         ' Stride limits
CenterStride   CON      750
LeftStride     CON      850

' -----[ Variables ]-----

MoveLoop       VAR      Nib         ' Loop for repeat movements
Pulses         VAR      Word        ' Pulse variable

CurrentTilt    VAR      Word
CurrentStride  VAR      Word
NewValue       VAR      Word

Dx             VAR      Pulses
Mx             VAR      Word

' -----[ EEPROM Data ]-----
' Take three full steps.
'
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL             CON      0
TC             CON      1
TR             CON      2

SL             CON      3

```



```

SC          CON      4
SR          CON      5

xx          CON      255
WalkForward DATA    TR, SL, TL, SR, xx
WalkBackward DATA   TR, SR, TL, SL, xx
TurnLeft   DATA    TL, SR, TC, SL, xx
FinishForward DATA   TR, SC, TC, xx

' -----[ Main Routine ]-----
'
Main_Program:
  GOSUB ResetCC

  FOR MoveLoop = 1 to 3
    Mx = WalkForward
    GOSUB Movement
  NEXT

  FOR MoveLoop = 1 to 3
    Mx = TurnLeft
    GOSUB Movement
  NEXT

  FOR MoveLoop = 1 to 3
    Mx = WalkBackward
    GOSUB Movement
  NEXT

  Mx = FinishForward
  GOSUB Movement
END

' -----[ Subroutines ]-----
' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = table index, table ends in xx

Movement:
  READ Mx, Dx          ' read next action
  Mx = Mx + 1

  IF Dx = xx THEN MovementDone ' skip if end of list

  GOSUB DoMovement    ' execute movement
  GOTO Movement       ' loop until done

DoMovement:
  BRANCH Dx, [TiltLeft, TiltCenter, TiltRight, StrideLeft,

```

```

        StrideCenter,StrideRight]
                                ' fall through if invalid
                                ' index
MovementDone:
    RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
    NewValue = LeftTilt
    GOTO MovementTilt

TiltCenter:
    NewValue = CenterTilt
    GOTO MovementTilt

TiltRight:
    NewValue = RightTilt

MovementTilt:
    FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
        PULSOUT TiltServo, Pulses
        PULSOUT StrideServo, CurrentStride
        PAUSE MoveDelay
    NEXT

    CurrentTilt = NewValue
    RETURN

StrideLeft:
    NewValue = LeftStride
    GOTO MovementStride

StrideCenter:
    NewValue = CenterStride
    GOTO MovementStride

StrideRight:
    NewValue = RightStride

MovementStride:
    FOR Pulses = CurrentStride TO NewValue STEP StrideStep
        PULSOUT TiltServo, CurrentTilt
        PULSOUT StrideServo, Pulses
        PAUSE MoveDelay
    NEXT

    CurrentStride = NewValue
    RETURN

```

```
' ----- Move feet to initial center position -----
ResetCC:
  CurrentTilt   = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN
```

Ok, we cheated. There is a little turning done in the program and that will be covered in more detail in the next experiment. The reason for including it here is that the pivot action is not explicitly done by a routine. Instead the general movement routine handles all actions. The pivot action is defined within the `TurnLeft` table.

The variable table handling is also changed to allow access to the six action functions like `TiltLeft`. Each is implemented as a routine that exits via a `RETURN` instruction. This is why the `DoMovement` routine is called via `GOSUB` rather than putting the `BRANCH` statement within a loop.

Also note the refinement with the `LOOKUP` statement. The named constant `xx` is used to terminate a variable length table. The value 255 is out of range for the values used and it allows the first value to be 0.

This approach will be the one used in subsequent programs in the book. Extending tables like `WalkForward` should be significantly easier since there are only six valid values. It is also possible to conserve space by using only 4 bits to store each value but this makes table definition and extraction very difficult due to limitations of PBASIC. Still, it is an option should a program become code-space constrained.

### Your Turn

- √ Run the example code.

## CHALLENGES

- √ Increase the walking speed and determine the maximum speed before the Toddler falls over.
- √ Determine whether the Toddler operates the same on different surfaces such as carpet, wood and tile.
- √ The Toddler can start moving its left or right foot first. Try changing the programs so that it moves the opposite foot first.
- √ Have the Toddler perform a little dance using a more complex series of steps such as moving the foot in the air forward and backward a few times.

## Chapter #3: Turning Around

---

### SLIP SLIDING TURNS

The Toddler is a bit stiff. It can only move its feet forward and backward but it cannot turn its feet relative to its body. This doesn't stop it from being able to turn. While walking in a straight line for the Toddler is somewhat similar to a person, turning is very different. The closest thing to turning like the Toddler for a person is trying to turn on ice with flat shoes.

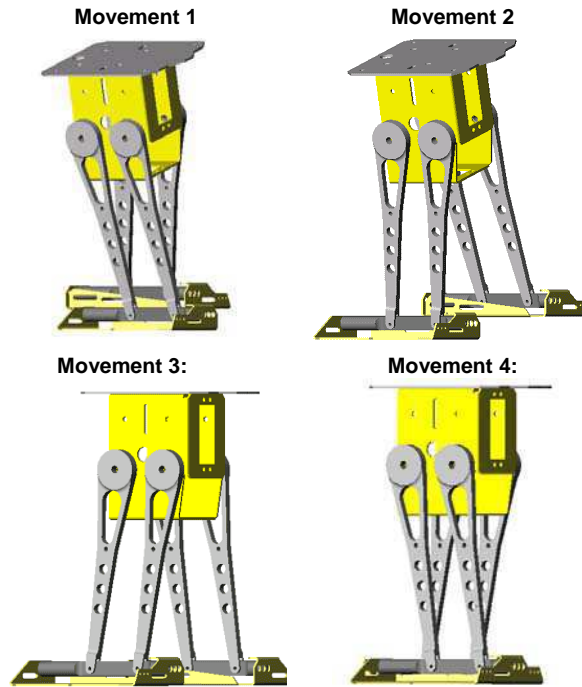
The process of turning right on ice is relatively simple. Put your left foot forward and place it on the ground. Pull the left foot towards you. Pull it back and you pivot to the right. If the ice is wet and slippery then it may take a number of attempts to turn 90 degrees. Put the right foot forward to turn left.

The standard Toddler does not do well on ice but it uses the same principle on other surfaces. The Toddler's feet are smooth metal that provides the slick surface. Turning works best when the surface the Toddler stands on provides some friction. If the surface is too slippery then it is possible to modify the Toddler's feet to provide more friction. This is typically done using tape that has a rougher surface than the Toddler's metal feet. The entire foot need not be covered. There only needs to be enough coverage to add friction for the turn. The tape should not have an affect on straight-line movement.

### ACTIVITY #1: MAKING A TURN

The Toddler turns by placing both feet flat on the ground and sliding them in opposite directions. Moving the feet in opposite directions is somewhat counterproductive because some of the forces are in opposite directions. The actual movement is more of a pivot than a turn.

The Toddler pivots only a small fraction of a circle at a time. It may take five to ten movements to turn 90 degrees (unless you put a small piece of grip tape on the feet), and twice that to turn around. The basic turning process is four movements like those shown in Figure 3-1. A left turn works the same way except the tilt and leg movements are reversed.



**Figure 3-1:** Walking in a Circle (right hand turns)

The first three movements place the left foot in front while the pivoting action occurs in the last movement. The following program performs this process multiple times. It performs both a right and a left turn. The key addition to Program 2.4 is the **TurnRight** table entry. The next section addresses two other entries that are presented in this program. These are **WideTurnLeft** and **PivotRight**.

```

' -----[ Title ]-----
' Toddler Program 3.1: Turning
' Demonstrates basic turning process which requires four movements
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      CON    13          ' Tilt servo on P12
StrideServo    CON    12          ' Stride servo on P13

' -----[ Constants ]-----

MoveDelay      CON    15          ' in microseconds
TiltStep       CON    5           ' TiltServo step size
StrideStep     CON    5           ' StrideServo step size

RightTilt      CON    620         ' Tilt limits
CenterTilt     CON    750
LeftTilt       CON    880

RightStride    CON    650         ' Stride limits
CenterStride   CON    750
LeftStride     CON    850

' -----[ Variables ]-----

MoveLoop       VAR    Nib         ' Loop for repeat movements
Pulses         VAR    Word        ' Pulse variable

CurrentTilt    VAR    Word
CurrentStride  VAR    Word
NewValue       VAR    Word

Dx             VAR    Pulses
Mx             VAR    Word

' -----[ EEPROM Data ]-----
'
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL             CON    0
TC             CON    1
TR             CON    2

SL             CON    3
SC             CON    4
SR             CON    5

```

```

xx                CON      255

WalkForward      DATA    TR, SL, TL, SR, xx
WalkBackward    DATA    TR, SR, TL, SL, xx

TurnLeft        DATA    TL, SR, TC, SL, xx
WideTurnLeft    DATA    TL, SR, TC, SL, TR, SL, TL, SR, xx

TurnRight       DATA    TR, SL, TC, SR, xx
PivotRight      DATA    TR, SL, TC, SR, TL, SL, TC, SR, xx

FinishForward   DATA    TR, SC, TC, xx

' -----[ Main Routine ]-----
Main_Program:
  GOSUB ResetCC

  FOR MoveLoop = 1 to 5
    Mx = TurnRight
    GOSUB Movement
  NEXT

  FOR MoveLoop = 1 to 5
    Mx = TurnLeft
    GOSUB Movement
  NEXT

  FOR MoveLoop = 1 to 5
    Mx = PivotRight
    GOSUB Movement
  NEXT

  FOR MoveLoop = 1 to 5
    Mx = WideTurnLeft
    GOSUB Movement
  NEXT

  Mx = FinishForward
  GOSUB Movement
END

' -----[ Subroutines ]-----
' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = table index, table ends in xx

Movement:

```



```

READ Mx, Dx                                ' read next action
Mx = Mx + 1

IF Dx = xx THEN MovementDone              ' skip if end of list

GOSUB DoMovement                          ' execute movement
GOTO Movement                             ' loop until done

DoMovement:
  BRANCH Dx, [TiltLeft, TiltCenter, TiltRight, StrideLeft,
             StrideCenter, StrideRight]
                                           ' will fall through if
                                           ' invalid index

MovementDone:
  RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
  NewValue = LeftTilt
  GOTO MovementTilt

TiltCenter:
  NewValue = CenterTilt
  GOTO MovementTilt

TiltRight:
  NewValue = RightTilt

MovementTilt:
  FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, CurrentStride
    PAUSE MoveDelay
  NEXT

  CurrentTilt = NewValue
  RETURN

StrideLeft:
  NewValue = LeftStride
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  GOTO MovementStride

StrideRight:
  NewValue = RightStride

```

```
MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN
```

### **Your Turn**

- √ Run the example code.
- √ Modify the code so the Toddler first pivots left, then right.

## ACTIVITY #2: DIFFERENT TURNS

The basic turns will get the Toddler where it wants to go but there are many variations on this theme. For example, turning in place can be useful in tight places. The `PivotRight` table entry shows how this can be done. In this case pivoting is accomplished by performing one turning movement by moving the leg forward. This is immediately followed with the same type of movement but it starts by stepping backward first. The combination results in two turn actions and a new orientation while leaving the Toddler in approximately the same position.

The other table entry included in the program is `wideTurnLeft`. This takes the Toddler around a circle with a wider radius. The trick is adding a forward step after each turning action. A close look at the Toddler's path will show that the movement is not really an arc but rather the perimeter of a polygon with rounded corners. Still, this is close enough to an arc that most people will think the Toddler is going around a circle.

Hopefully the simplicity of the state transition approach is apparent with Program 3.1. It is identical to Program 2.4 except for the additional tables and calls that utilize these tables.

### Your Turn

- √ Modify Program 3.1 to execute different styles of turns.

## CHALLENGES

- √ Only a few turning variations were presented in the sample programs. Add the table entries needed to perform the actions not included.
- √ The Toddler is symmetrical in construction and movement. It can make a turn going forwards or backwards. Create a program that can perform the actions presented but going backwards instead.
- √ Full leg movements were used in the sample applications. Determine what happens if the movements are shorter. For example, instead of moving from state **CR** to **CL**, try moving from **CR** to **CC**.
- √ **WideTurnLeft** turns the Toddler to the left but the turn radius is wider than **TurnLeft**. Make the Toddler's turn even wider. Hint: there are two ways of doing this. One is related to forward movement. The other is related to turn movement.

## Chapter #4: Coordinated Walking

---

### MULTIPLE TABLES FOR MANY ACTIONS

Walking and turning are useful operations but they don't get the Toddler very far. It is possible to string together a number of actions using multiple `GOSUB` statements but this can get tedious. It is also less efficient than the approach presented in this experiment.

The program in Chapter #3 used tables to store a series of basic movements. More complex actions can be done using very long tables but an alternative is to utilize these tables from a higher-level table. Instead of indicating whether the Toddler leans left or right, a movement table will include actions such as turn right, walk backward 10 steps, pivot left and walk forward 10 steps.

The second activity in this experiment uses this approach to move the Toddler in more complex paths than the earlier experiments, but first we take a look at how to determine if a table is part of one set or another. This will allow the `Movement` routine to determine whether a table is a basic set of movement commands or if the table contains more complex commands. The `Movement` routine can then process the commands accordingly.

### ACTIVITY #1: WHICH TABLE?

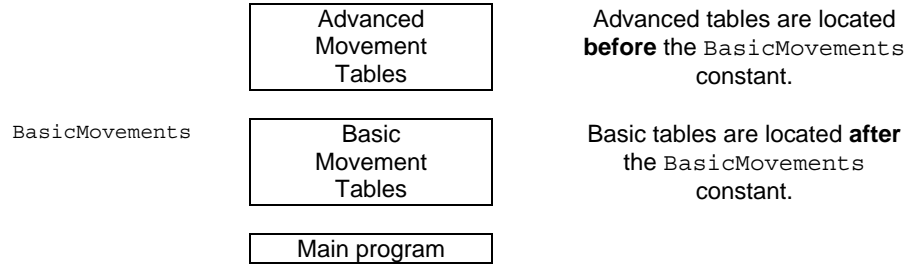
There are advantages to using high-level actions with the Toddler such as making a left turn versus low-level actions such as leaning left and moving the left foot forward. Both are needed and prior examples have shown how low level actions can be combined in tables to provide a higher level of abstraction. Taking this approach to the next level requires a different set of tables whose elements reference the lower level tables.

Using two types of tables is possible using two different routines but there is an advantage to using a single routine for both. This allows a program to freely mix the use of these two types of tables in the program. The program in this activity shows how this can be done. The approach is then used in the next activity's program.

This program does not make the Toddler walk but it does use the BASIC Stamp to run the program. It uses the BASIC Stamp Editor's Debug Terminal to display the output generated by `DEBUG` statements in the program. Most programmers familiar with the BASIC Stamp will already know about the `DEBUG` statement but check out the [BASIC](#)

Stamp Manual if you are not. Also, the serial cable will remain connected to the Toddler for this experiment.

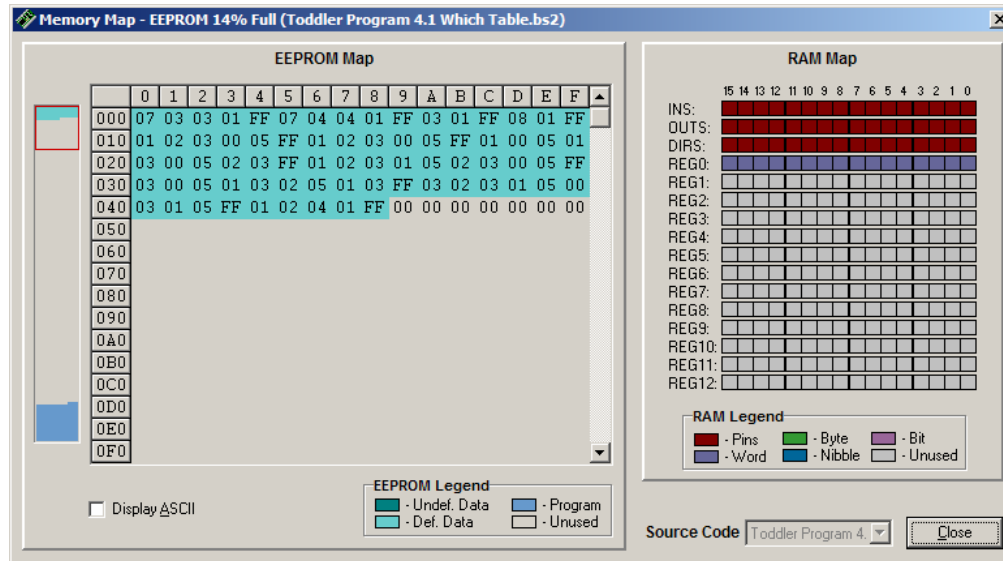
This program assumes that two kinds of tables will be used with the program and that each type of table will be in its own area of the program memory. There is no restriction that tables be adjacent, only that they be above or below the address boundary that is designated by the `BasicMovements` constant declaration.



**Table 4-1:** Coordinated Walking Table Structure

The program uses a `Movement` routine that has the index of the table in the `Mx` variable. It uses the `DEBUG` statement to output a string in the Debug Terminal on the PC that indicates whether index is for an advanced or basic movement. In the next activity, the `Movement` routine will be replaced by one that actually interprets the tables to make the Toddler move.

Understanding how data is stored in the BASIC Stamp's EEPROM is very helpful. If you choose Run → Memory Map you will see that data builds from EEPROM address 0 towards 2048. On the other hand, your PBASIC program is stored from location 2048 and builds downward to location 0. This is shown graphically in Figure 4-1.



**Figure 4-1:** EEPROM Memory Map for “Which Table” Program

Note that the **DATA** statement stores data from location 0 and the PBASIC program builds from location 2048 towards 0. Slide the yellow box in the “Condensed EEPROM Map” down to change the viewing area in the “Detailed EEPROM Map” so you can see your PBASIC program. Figure 4-2 shows the **DEBUG** output from the program.

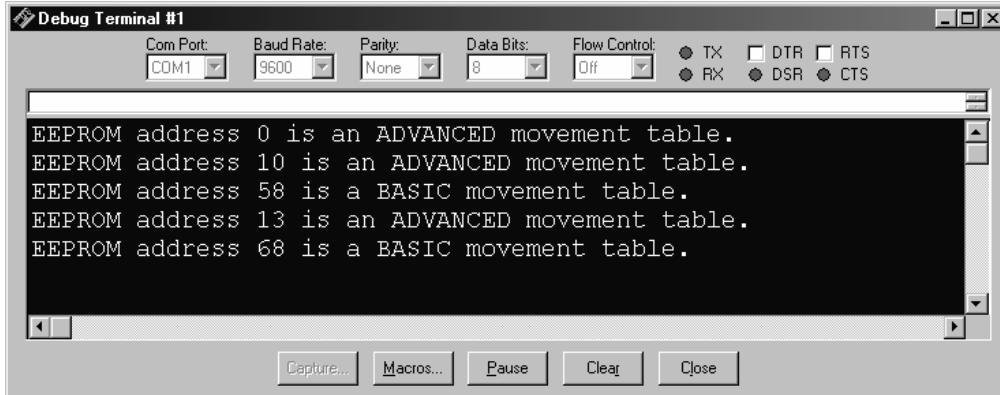


Figure 4-2: Debug Terminal shows EEPROM pointer Mx.

```
' -----[ Title ]-----
' Toddler Program 4.1: Which Table
' Demonstrates basic and advanced tables
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Constants ]-----

' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL          CON    0
TC          CON    1
TR          CON    2

SL          CON    3
SC          CON    4
SR          CON    5

xx          CON    255          ' Movement table limit
                                   ' entry may be increased.

bFinish     CON    0
bForward    CON    1
bBackward   CON    2
bLeftTurn   CON    3
bRightTurn  CON    4
bPivotLeft  CON    5
bPivotRight CON    6
```



```

' -----[ Variables ]-----
Mx          VAR      Word

' -----[ EEPROM Data ]-----
'
' ---- Advanced Movement Tables ----
' Used for

LeftSemicircle  DATA    7, bLeftTurn,  bLeftTurn,  bForward, xx
RightSemicircle DATA    7, bRightTurn, bRightTurn, bForward, xx

WalkForward3    DATA    3, bForward,  xx
WalkForward8    DATA    8, bForward,  xx

' ---- Basic Movement Tables ----
' These tables can contain movement support codes used in the advanced
' movement tables.

BasicMovements CON      Forward

Forward         DATA    1, TR, SL, TL, SR, xx
Backward        DATA    1, TR, SL, TL, SR, xx
LeftTurn        DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn       DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft       DATA    3, TL, SR, TC, SL, TR, SR, TC, SL, xx
PivotRight      DATA    3, TR, SL, TC, SR, TL, SL, TC, SR, xx

Finish          DATA    1, TR, SC, TC, xx

' -----[ Main Code ]-----
' Walking routine is defined by the Mx value. Mix the basic and advanced
' movement tables to develop the walking routines for your Toddler.

Main_Program:
  Mx = LeftSemicircle
  GOSUB Movement

  Mx = WalkForward3
  GOSUB Movement

  Mx = PivotRight
  GOSUB Movement

  Mx = WalkForward8
  GOSUB Movement

  Mx = Finish

```

```

GOSUB Movement
END

' ----- [ Subroutines ]-----

Movement:
  IF Mx >= BasicMovements THEN BasicMovementTable

  DEBUG "EEPROM address ", dec Mx, " is an ADVANCED movement table.", cr
  RETURN

BasicMovementTable:
  DEBUG "EEPROM address ", dec Mx, " is a BASIC movement table.", cr
  RETURN

```

This approach to memory partitioning takes into account that PBASIC allocates space for **DATA** statements in the order that they appear in the program. Not all programming languages do this so the technique should only be applied when the appropriate support is available. It is handy in this instance because other approaches to differentiating data are more cumbersome. This approach can be prone to programmer errors if the tables are not grouped properly but keeping the tables relatively close to each other in the program text makes it easy to spot such problems.

### **Your Turn**

- √ Run the example code.
- √ Understand how the EEPROM address displayed in the Debug Terminal correlates to an advanced or basic movement table.

## ACTIVITY #2: FIGURE 8S AND SQUARE DANCING

Now we take a look at making the Toddler execute more complex movements using the dual table types presented in Activity #1. One set of tables handles low-level actions such as tilting and leg movements. The second set of tables handles higher-level actions such as turning a corner and walking in a large circle.

In this Activity, the program makes the Toddler walk in a Figure 8 and a large square. The use of a high-level action table allows easy creation of more complex movement sequences. In this case, the use of higher level sequences like `LeftSemicircle` cause the Toddler to execute a large number of basic foot movements.

4

The program implements a more sophisticated version of the `Movement` routine than found in prior Activities. In this case, the `Mx` variable can contain an index for either a basic or an advanced table. The `Movement` routine will execute the appropriate table. The decoding process is a bit complex so we have included `DEBUG` statements to help present the execution process. The `DEBUG` statements are actually comments in the listing but they can be changed by doing a “Replace All” in the editor from “`'DEBUG`” to “`DEBUG`”. The converse will change the lines back to comments.

The `DEBUG` statements are only useful when the Toddler is connected to the PC since the display of information is done on the PC. While it is possible to keep the Toddler connected to the PC while it is walking, these more advanced movement sequences move the Toddler in large areas. A laptop or a long serial cable may be necessary to handle these larger movement areas.

What we have found useful is to instead use the `DEBUG` version while running the Toddler with the power switch in the download mode. In this case, the Toddler’s servos do not move but the program continues to execute. The `DEBUG` statements show what the Toddler would be doing if the power switch was in the RUN mode (Position 2).

When running in the download mode, the Toddler will continue to send pulses to the servos even though the servos receive no power and hence do not rotate. The delay does slow down debug presentation though. It is possible to add a `RETURN` statement immediately after the `DoMovement` label to eliminate this thereby making the debug process go faster. Just make sure to comment out or remove the `RETURN` statement or the Toddler will not move. Now for the code.

```

' -----[ Title ]-----
' Toddler Program 4.2: Advanced Walking
' Demonstrates the use of basic and advanced tables for figure 8s
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      CON      13          ' Tilt servo on P12
StrideServo    CON      12          ' Stride servo on P13
LineFeed       CON      10          ' Line Feed

' -----[ Constants ]-----

MoveDelay      CON      18          ' in milliseconds
TiltStep        CON      5          ' TiltServo step size
RightTilt       CON      620        ' Tilt limits was 620
CenterTilt      CON      750        ' was 750
LeftTilt        CON      880        ' was 880

StrideStep      CON      5          ' StrideServo step size
RightStride     CON      650        ' Stride limits was 650
CenterStride    CON      750        ' was 750
LeftStride      CON      850        ' was 850

' -----[ Variables ]-----

FigureLoop     VAR      Nib
MoveLoop        VAR      Byte      ' Loop for repeat movements
MoveLoopLimit  VAR      Byte

SubMoveLoop     VAR      Byte      ' Loop for repeat submovements
SubMoveLoopLmt VAR      Byte

Pulses          VAR      Word      ' Pulse variable

CurrentTilt     VAR      Word
CurrentStride   VAR      Word
NewValue        VAR      Word

Dx              VAR      Pulses

Mx              VAR      Word
MxCurrent       VAR      Word

Sx              VAR      Word
SxCurrent       VAR      Word

```

```

' -----[ EEPROM Data ]-----
'
' -----Movement Support Codes
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL          CON      0
TC          CON      1
TR          CON      2

SL          CON      3
SC          CON      4
SR          CON      5

xx          CON      255

' ----- Movement Value Tables -----
'
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
'
' Note: ALL movement tables must be in this section

LeftSemicircle DATA  7, bLeftTurn, bLeftTurn, bForward, xx
RightSemicircle DATA 7, bRightTurn, bRightTurn, bForward, xx

WalkForward3  DATA  3, bForward, xx
WalkForward8  DATA  8, bForward, xx
WalkForward1  DATA  1, bForward, xx
WalkBackward8 DATA  8, bBackward, xx
PivotLeft4   DATA  4, bPivotLeft, bForward, xx

' ----- Basic Movement Codes -----
'
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish      CON      0
bForward     CON      1
bBackward    CON      2
bLeftTurn    CON      3
bRightTurn   CON      4
bPivotLeft   CON      5
bPivotRight  CON      6

' ----- Basic Movement Tables -----
'

```

```
' These tables can contain Movement Support Codes.

BasicMovements  CON      Forward

Forward          DATA    1, TR, SL, TL, SR, xx
Backward         DATA    1, TR, SR, TL, SL, xx

LeftTurn         DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn        DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft        DATA    3, TL, SR, TC, SL, TR, SR, TC, SL, xx
PivotRight       DATA    3, TR, SL, TC, SR, TL, SL, TC, SR, xx

Finish           DATA    1, TR, SC, TC, xx

GOSUB ResetCC

'DEBUG "Forward = ", HEX Forward, CR,LineFeed
'DEBUG "Backward = ", HEX Backward, CR,LineFeed
'DEBUG "LeftTurn = ", HEX LeftTurn, CR,LineFeed
'DEBUG "RightTurn = ", HEX RightTurn, CR,LineFeed
'DEBUG "PivotLeft = ", HEX PivotLeft, CR,LineFeed
'DEBUG "PivotRight = ", HEX PivotRight, CR,LineFeed
'DEBUG "LeftSemicircle = ", HEX LeftSemicircle, CR,LineFeed
'DEBUG "RightSemicircle = ", HEX RightSemicircle, CR,LineFeed
'DEBUG "WalkForward3 = ", HEX WalkForward3, CR,LineFeed
'DEBUG "WalkForward8 = ", HEX WalkForward8, CR,LineFeed
'DEBUG "WalkForward1 = ", HEX WalkForward1, CR,LineFeed
'DEBUG "WalkBackward8 = ", HEX WalkBackward8, CR,LineFeed
'DEBUG "PivotLeft4 = ", HEX PivotLeft4, CR,LineFeed
'DEBUG "Finish = ", HEX Finish, CR,LineFeed

' -----[ Main Code ]-----

Main_Program:

  GOSUB ResetCC

  ' Make a Figure 8

  FOR FigureLoop = 1 TO 5
    Mx = LeftSemicircle
    GOSUB Movement

    Mx = WalkForward3
    GOSUB Movement

    Mx = RightSemicircle
    GOSUB Movement

    Mx = WalkForward3
```

```

    GOSUB Movement
NEXT

' Make a big polygon

FOR FigureLoop = 1 TO 5
  Mx = PivotRight
  GOSUB Movement

  Mx = WalkForward8
  GOSUB Movement
NEXT

Mx = Finish
GOSUB Movement
END

' -----[ Subroutines ]-----

' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = movement table index, table ends in xx
' or
'       Mx = submovement table index, table ends in xx
'
' Note: All submovement tables come after the movment tables in this file.

Movement:
  IF Mx < BasicMovements THEN SetupMovement
  'DEBUG cr,"use submovements",cr
  Sx = Mx                                ' setup for submovement table only
  GOSUB StartSubMovement                 ' one pass through submovement table
  RETURN

SetupMovement:
  READ Mx, MoveLoopLimit                 ' read movement table repeat count
  Mx = Mx + 1

StartMovement:
  FOR MoveLoop = 1 TO MoveLoopLimit
  'DEBUG cr,cr, hex (Mx-1), " Start Movement ", dec MoveLoop, " of "
  'DEBUG dec MoveLoopLimit,cr
  MxCurrent = Mx                         ' start of movements

MovementLoop:
  READ MxCurrent, Sx                     ' read next submovment byte
  MxCurrent = MxCurrent + 1
  'DEBUG cr, " SX = ", dec Sx, " movement"
  IF Sx = xx THEN MovementDone
  ' skip if end of list

```

```

LOOKUP Sx, [Finish,Forward,Backward,LeftTurn,RightTurn,
PivotLeft,PivotRight],Sx
' lookup submovement table index

GOSUB StartSubMovement
GOTO MovementLoop

MovementDone:
NEXT
RETURN

'-----

StartSubMovement:
READ Sx, SubMoveLoopLmt
' read submovement table repeat count

Sx = Sx + 1

FOR SubMoveLoop = 1 TO SubMoveLoopLmt
'DEBUG cr, "      SX = ", hex (Sx-1), " submovement ", dec SubMoveLoop
'DEBUG " of ", dec SubMoveLoopLmt, " "
SxCurrent = Sx

SubMovementLoop:
READ SxCurrent, Dx
SxCurrent = SxCurrent + 1
' read next submovement action

IF Dx = xx THEN SubMovementDone
' skip if end of list

GOSUB DoMovement
GOTO SubMovementLoop
' execute movement

SubMovementDone:
NEXT
RETURN

DoMovement:
BRANCH Dx, [TiltLeft,TiltCenter,TiltRight,StrideLeft,
StrideCenter,StrideRight]
' will fall through if invalid index

RETURN

' ---- Movement routines can be called directly as subroutines

TiltLeft:
NewValue = LeftTilt
'DEBUG "TL, "
GOTO MovementTilt

TiltCenter:
NewValue = CenterTilt
'DEBUG "TC, "

```



```

GOTO MovementTilt

TiltRight:
  NewValue = RightTilt
  'DEBUG "TR, "

MovementTilt:
  FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, CurrentStride
    PAUSE MoveDelay
  NEXT

  CurrentTilt = NewValue
  RETURN

StrideLeft:
  NewValue = LeftStride
  'DEBUG "SL, "
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  'DEBUG "SC, "
  GOTO MovementStride

StrideRight:
  NewValue = RightStride
  'DEBUG "SR, "

MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride
  'DEBUG "resetCC", CR, LineFeed

  FOR Pulses = 1 TO 500 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay

```

```
    NEXT  
DoReturn:  
    RETURN
```

The **Main\_Program** now executes so the Toddler performs two large movements: a figure 8 and a square. The **Movement** routine is called to execute high-level tables that include commands such as left turn. The **bLeftTurn** value is used in the table because the **DATA** statement is only storing bytes in this example. These values are used with a **LOOKUP** statement in the **Movement** routine to select the appropriate low level table to use for basic movements. One high-level table entry causes the Toddler to execute many low level movements.

The **Movement** routine still handles low-level tables using the technique outlined in Activity #1. For example, the **Finish** table uses only a few basic movements. If the **Movement** routine did not handle both types of tables then either a low level routine would have to be called or a high level table would have to be created.

**Your Turn**

- √ Run the example program.
- √ Modify the code so the Toddler executes the same pattern in a smaller geographic area.

## CHALLENGES

- √ A challenge in Chapter #4 asked to extend the movement routine so it could handle repetitions of subsections in a table as in.

```
SpecialMovement      DATA  4, TL, SR, TC, SL, xx
                    DATA  2. TR, SL, TL, SR, xx
                    DATA  2. TL, SR, TC, SL, xx
```

Provide this same type of facility for both types of movement tables.

- √ Implement the prior challenge. Then reduce the multiple calls to **Movement** shown below

```
Mx = LeftSemicircle
GOSUB Movement

Mx = WalkForward3
GOSUB Movement

Mx = RightSemicircle
GOSUB Movement

Mx = WalkForward3
GOSUB Movement
```

to the following

```
Mx = Figure8
GOSUB Movement
```

Hint: The Figure 8 table will be a combination of the four tables listed above.

- √ Generate a set of symmetrical useful high level and low level movement tables. High level movements might include **TurnAroundLeft**, **TurnAroundRight**, **WalkForward1Foot**, and **WalkBackward1Foot**. Low level movements might include **ReversePivotRight** and **ReversePivotLeft**.

## Chapter #5: Following Light

### BRINGING YOUR CREATURE TO LIFE

The photoresistors in your kit can be used to make your Toddler detect variations in light level. With some programming, your Toddler can be transformed into a photophile (a creature attracted to light), or a photophobe (a creature that tries to avoid light).

To sense the presence and intensity of light you'll build a couple of photoresistor circuits on your Toddler. A photoresistor is a light-dependent resistor (LDR) that covers the spectral sensitivity similar to that of the human eye. The active elements of these photoresistors are made of Cadmium Sulfide (CdS). Light enters into the semiconductor layer applied to a ceramic substrate and produces free charge carriers. A defined electrical resistance is produced that is inversely proportional to the illumination intensity. In other words, darkness produces high resistance, and high illumination produces very small amounts of resistance.

The specific photoresistors included in the Toddler kit are from EG&G Vactec (#VT935G). If you need additional photoresistors they are available from Parallax's Component Shop as well as from many electronic component suppliers. See Appendix A: Toddler Parts Lists and Sources. The specifications of these photoresistors are shown in Table 5-1.

Resistance (Ohms)					Peak Spectral Response nm	$V_{MAX}$	Response Time @ 1 fc (ms, typ.)	
10 Lux 2850K			Dark				Rise (1-1/e)	Fall (1/e)
Min	Typ.	Max.	Min.	Sec.				
20K	29.0K	38K	1M	10	550	100	35	5

**Table 5-1:** EG&G Vactec Photoresistor Specifications

Luminance is a scientific name for the measurement of incident light. The unit of measurement of luminance is commonly the "foot-candle" in the English system and the "lux" in the metric system. While using the photoresistors we won't be concerned about lux levels, just whether or not luminance is higher or lower in certain directions. The Toddler can be programmed to use the relative light intensity information to make

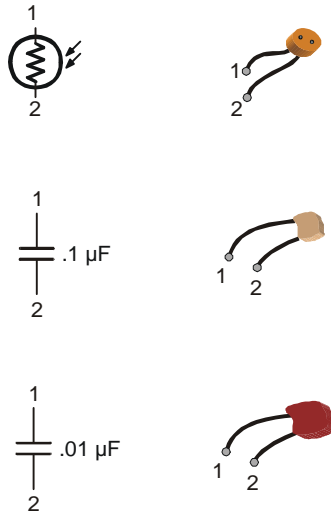
navigation decisions. For more information about light measurement with a microcontroller, take a look at the [Applied Sensors](#) experiment “Light on Earth and Data Logging”.

### ACTIVITY #1: BUILDING AND TESTING PHOTOSENSITIVE EYES

Figure 5-1 shows the capacitors and photoresistor used in this experiment along with their schematic symbols. Both capacitors are nonpolar, meaning that terminals 1 and 2 as shown may be swapped without affecting the circuit. In addition to the capacitors, you’ll also need two (2) 220 ohm resistors (color code red-red-brown).

#### Parts Required

- (2) Photoresistors
- (2) 0.1  $\mu\text{F}$  capacitors
- (2) 0.01  $\mu\text{F}$  capacitors
- (2) 220 ohm resistors (not pictured)
- (misc.) jumper wires



**Figure 5-1:** Photoresistor and Capacitor Circuit Symbols and Parts

Figure 5-2 and Figure 5-3 show the resistor/capacitor (RC) circuit for each photoresistor. A photoresistor is an analog device. Its value varies continuously as luminance, another analog value, varies. The photoresistor's resistance is very low when its light-sensitive surface is placed in direct sunlight. As the light level decreases, the photoresistor's resistance increases. In complete darkness, the photoresistor's value can increase to more than 1 M Ohm. Even though the photoresistor is analog; its response to light is nonlinear. This means if the input source (luminance) varies at a constant rate, the photoresistor's value does not necessarily vary at a constant rate.

5

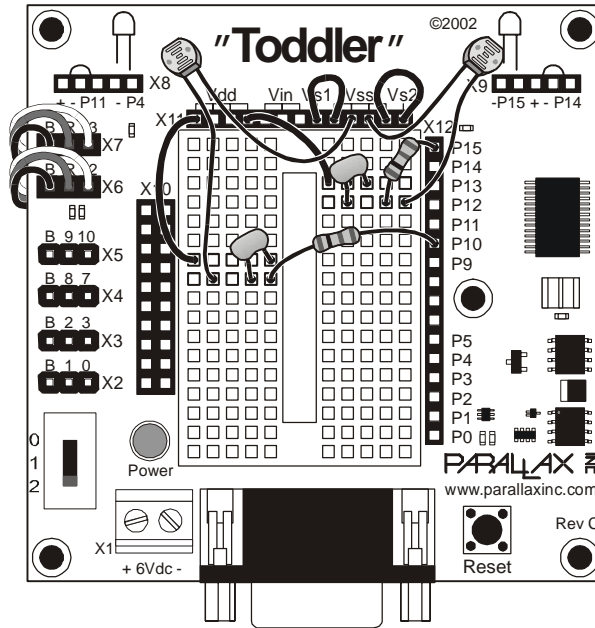


Figure 5-2: Light Following Pictorial

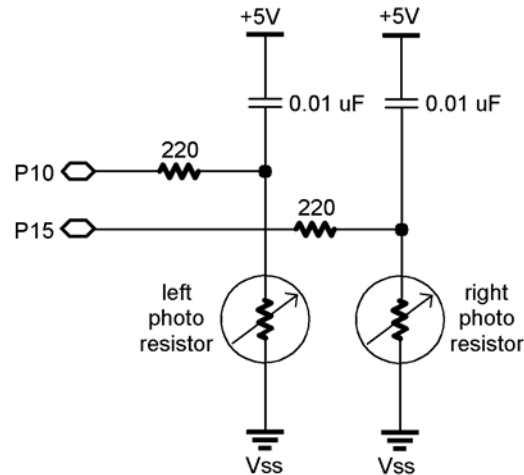


Figure 5-3: Light Following Schematic

### Programming to Measure the Resistance

The circuit in Figure 5-2 and Figure 5-3 was designed for use with the PBASIC `RCTIME` command. This command can be used with an RC circuit where one value, either R or C, varies while the other remains constant. The `RCTIME` command lends itself to measuring the variable values because it takes advantage of a time varying property of RC circuits.

For one of the RC circuits, the first step in setting up the `RCTIME` measurement is charging the lower plate of the capacitor to 5 V. Setting the I/O pin connected to the lower capacitor plate by the 220 ohm resistor high for a few ms takes care of this. Next, the `RCTIME` command can be used to take the measurement of the time it takes the lower plate to discharge from 5 to 1.4 V. Why 1.4 V? Because that's the BASIC Stamp I/O pin's threshold voltage. When the voltage at an I/O pin set to input is above 1.4 V, the value in the input register bit connected to that I/O pin is "1." When the voltage is below 1.4 V, the value in the input register bit is "0."

In this circuit `RCTIME` measures the time it takes the voltage at the lower plate of the capacitor in one of the photoresistor circuits to drop from 5 to 1.4 V. This discharge is directly proportional to the photoresistor's resistance. Since this resistance varies with



luminance (exposure to varying levels of light), so does the time. By measuring this time, relative light exposure can be inferred. See the [BASIC Stamp Manual](#) for a detailed discussion of **RCTIME**.

The **RCTIME** command changes the I/O pin from output to input. As soon as the I/O pin becomes an input, the voltage at the lower plate of the capacitor starts to fall according to the time equation just discussed. The BASIC Stamp starts counting in 2 us increments until the voltage at the capacitor's lower plate drops below 1.4 V.



**Photoresistors work best** in lower light levels while seeking natural light. Direct sunlight is too bright for the photoresistor circuits.

**5**

Run Program Listing 5.1. It demonstrates how to use the **RCTIME** command to read the photoresistors. This program makes use of the Debug Terminal, so leave the serial cable connected to the Toddler board while the program is running.

```

' -----[ Title ]-----
' Toddler Program 5.1: Photoresistor RCTime Display
' Displays the R/C discharge time for two photoresistor circuits
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
LPhotoCircuit  CON    10
RPhotoCircuit  CON    14

' -----[ Variables ]-----
LPhotoVal      VAR    Word           ' Stores measured RC times
RPhotoVal      VAR    Word           ' of photoresistors.

' -----[ Initialization ]-----
DEBUG CLS                                     ' Open, clear Debug window

' -----[ Main Code ]-----
DO
' Measure RC time for left photoresistor.
HIGH LPhotoCircuit                          ' Set to output-high.
PAUSE 3                                       ' Pause for 3 ms.
RCTIME LPhotoCircuit,1,LPhotoVal            ' Measure R/C time on left

' Measure RC time for right photoresistor.
HIGH RPhotoCircuit                          ' Set to output-high.
PAUSE 3                                       ' Pause for 3 ms
RCTIME RPhotoCircuit,1,RPhotoVal           ' Measure R/C time on right

' Display RC time measurements using Debug Terminal.

DEBUG HOME, "Left = ", DEC5 LPhotoVal, " Right = ", DEC5 RPhotoVal, cr
LOOP

```

### How The Photoresistor Display Works

Two word variables, `RPhotoVal` and `LPhotoVal` are declared for storing the RC time values of the left and right photoresistors. The main routine then measures and displays the RC times for each RC circuit. The code for reading the right RC circuit is shown below. First, the I/O pin `RPhotoCircuit` is set to output-high. Next, a 3 ms pause allows enough time for the capacitor to charge. After 3 ms, the lower plate of the capacitor is close enough to 5 V and is ready for the RC time measurement. The `RCTIME` command measures the RC time on I/O pin `RPhotoCircuit`, with a beginning state of “1” (5 V), and stores the result in the `RPhotoVal` variable. Remember, the value stored in `RPhotoVal` is a number. This number tells how many 2 us increments passed before the voltage at the lower plate of the capacitor passed below the I/O pin’s 1.4 V threshold.

```
HIGH RPhotoCircuit           ' Set to output-high.
PAUSE 3                      ' Pause for 3 ms
RCTIME RPhotoCircuit,1,RPhotoVal  ' Measure R/C time on right
```

### Your Turn

- √ Try replacing one of the 0.01 μF capacitors with a 0.1 μF capacitor. Which circuit fares better in bright light, the one with the larger (0.1 μF) or the one with the smaller (0.01 μF) capacitor? What is the effect as the surroundings get darker and darker? Do you notice any symptoms that would indicate that one or the other capacitor would work better in a darker environment?

## ACTIVITY #2: A LIGHT COMPASS

If you focus a flashlight beam in front of the Toddler, the circuit and programming techniques just discussed can be used to make the Toddler turn so that it's pointing at the flashlight beam. Make sure the photoresistors are pointed so that they can make a light comparison. Aside from each being pointed 45° outward from the center-line of the Toddler, they also should be oriented so they are pointing 45° downward from horizontal. In other words, point the faces of the photoresistors down toward the table top. Then, use a bright flashlight to make the Toddler track the direction of the light.

### Programming the Toddler to Point at the Light

Getting the Toddler to track a light source is a matter of programming it to compare the value measured at each photoresistor. Remember that as the light gets dimmer, the photoresistor's value increases. So, if the photoresistor value on the right is larger than that of the photoresistor on the left, it means it's brighter on the left. Given this situation, the Toddler should turn left. On the other hand, if the **RCTIME** of the photoresistor on the left is larger than that of the photoresistor on the right, the right side is brighter and the Toddler should turn right.

To keep the Toddler from changing directions too often, a parameter for **DeadBand** is introduced. **DeadBand** is a range of values wherein the system makes no attempt at correction. If the numbers go above or below the **DeadBand**, then the system corrects accordingly. The most convenient way to measure for **DeadBand** is to subtract the left **RCTIME** from the right **RCTIME**, or visa versa, then take the absolute value. If this absolute value is within the **DeadBand** limits, then do nothing; if otherwise, program an appropriate adjustment.

```

' -----[ Title ]-----
' Toddler Program 5.2: Light Compass
' Points towards the most well-lit area in the room
' Adjust DeadBand for Toddler's sensitivity to light levels
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      CON    13      ' Tilt servo on P13
StrideServo    CON    12      ' Stride servo P12
LPhotoCircuit  CON    10
RPhotoCircuit  CON    15

' -----[ Constants ]-----

MoveDelay      CON    15      ' in microseconds
TiltStep       CON    5       ' TiltServo step size
StrideStep     CON    5       ' StrideServo step size

RightTilt      CON    620     ' Tilt limits
CenterTilt     CON    750
LeftTilt       CON    880

RightStride    CON    650     ' Stride limits
CenterStride   CON    750
LeftStride     CON    850

DeadBand       CON    30      ' Light sensitivity diff

' -----[ Variables ]-----

LPhotoVal      VAR    word     ' Stores measured R/C times
RPhotoVal      VAR    word     ' of photoresistors

FigureLoop     VAR    Nib
MoveLoop       VAR    Byte     ' Loop for repeat movements
MoveLoopLimit  VAR    Byte

SubMoveLoop    VAR    Byte     ' Loop for repeat
SubMoveLoopLmt VAR    Byte     ' submovements

Pulses         VAR    Word     ' Pulse variable

CurrentTilt    VAR    Word
CurrentStride  VAR    Word
NewValue       VAR    Word

Dx             VAR    Pulses
Mx             VAR    Word

```

```

MxCurrent      VAR      Word
Sx             VAR      Word
SxCurrent      VAR      Word

' -----[ EEPROM Data ]-----

' ----- Movement Support Codes -----
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL             CON      0
TC             CON      1
TR             CON      2

SL             CON      3
SC             CON      4
SR             CON      5

xx             CON      255

' ----- Movement Value Tables -----
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
' Note: ALL movement tables must be in this section

LeftSemicircle DATA    7, bLeftTurn, bLeftTurn, bForward, xx
RightSemicircle DATA    7, bRightTurn, bRightTurn, bForward, xx

WalkForward3   DATA    3, bForward, xx
WalkForward8   DATA    8, bForward, xx

' ----- Basic Movement Codes -----
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish        CON      0
bForward       CON      1
bBackward      CON      2
bLeftTurn      CON      3
bRightTurn     CON      4
bPivotLeft     CON      5
bPivotRight    CON      6

' ----- Basic Movement Tables -----
' These tables can contain Movement Support Codes.

BasicMovements CON      Forward

Forward        DATA    1, TR, SL, TL, SR, xx

```

```

Backward      DATA    1, TR, SR, TL, SL, xx
LeftTurn     DATA    1, TL, SR, TC, SL, xx
RightTurn    DATA    1, TR, SL, TC, SR, xx

PivotLeft    DATA    3, TL, SR, TC, SL, xx
PivotRight   DATA    3, TR, SL, TC, SR, xx

Finish       DATA    1, TR, SC, TC, xx

' -----[ Initialization ]-----
GOSUB ResetCC           ' Initialize feet

' -----[ Main Code ]-----
Main:

' Measure RC time for left photoresistor.
HIGH LPhotoCircuit     ' Set to output-high.
PAUSE 3                 ' Pause for 3 ms.
RCTYPE LPhotoCircuit,1,LPhotoVal ' Measure R/C time on left

' Measure RC time for right photoresistor.
HIGH RPhotoCircuit     ' Set to output-high.
PAUSE 3                 ' Pause for 3 ms
RCTYPE RPhotoCircuit,1,RPhotoVal ' Measure R/C time on right

' Measure difference between RPhotoVal and LPhotoVal, decide what to do
DEBUG home, "Left = ", dec LPhotoVal, " Right = ",dec RPhotoVal,cr

IF ABS(LPhotoVal-RPhotoVal) < DeadBand THEN main
IF LPhotoVal > RPhotoVal THEN turn_right
IF LPhotoVal < RPhotoVal THEN turn_left

'----- Navigation Routines -----

Turn_left:              ' turn left towards light
Mx = PivotLeft
GOSUB Movement
GOTO main                ' go back to main routine.

Turn_right:             ' turn right towards light
Mx = PivotRight
GOSUB Movement
GOTO main                ' go back to main routine.

' -----[ Subroutines ]-----

' ----- Movement: Move feet using DATA table referenced by Mx -----

```

```

' Input: Mx = movement table index, table ends in xx
'   or
'       Mx = submovement table index, table ends in xx
'
' Note: All submovement tables come after the movement tables in this file.

Movement:
  IF Mx < BasicMovements THEN SetupMovement

  MxCurrent = Mx           ' setup to use submovement
  MoveLoopLimit = 1       ' table
  GOTO StartMovement

SetupMovement:
  READ Mx, MoveLoopLimit  ' read movement table
  MxCurrent = Mx + 1      ' repeat count

StartMovement:
  FOR MoveLoop = 1 to MoveLoopLimit
  Mx = MxCurrent          ' Mx = start of movement
                          ' table

  'DEBUG DEC Mx, " Movement ", DEC MoveLoop, " of ", DEC MoveLoopLimit,CR

  IF Mx < BasicMovements THEN MovementLoop
                          ' skip if movement table
  SxCurrent = Mx          ' SxCurrent = submovement table index
  GOTO StartSubMovement  ' enter middle of loop

MovementLoop:
  READ Mx, SxCurrent      ' read next submovement byte
  Mx = Mx + 1
  IF SxCurrent = xx THEN MovementDone
                          ' skip if end of list
  'DEBUG " ", DEC SxCurrent, " movement",CR
  LOOKUP SxCurrent, [Finish,Forward,Backward,LeftTurn,
  RightTurn,PivotLeft,PivotRight],SxCurrent
  ' lookup submovement table index
StartSubMovement:
  READ SxCurrent, SubMoveLoopLmt
                          ' read submovement table

  SxCurrent = SxCurrent + 1

  FOR SubMoveLoop = 1 TO SubMoveLoopLmt
  Sx = SxCurrent
  'DEBUG " ", DEC Sx, " submovement "
  'DEBUG DEC SubMoveLoop, " of ", DEC SubMoveLoopLmt,CR

SubMovementLoop:
  READ Sx, Dx            ' read next submovement
  Sx = Sx + 1           ' action

```



```

IF Dx = xx THEN SubMovementDone
                                ' skip if end of list
GOSUB DoMovement                ' execute movement
GOTO SubMovementLoop

SubMovementDone:
NEXT
IF Mx < BasicMovements THEN MovementLoop

MovementDone:
NEXT
RETURN

DoMovement:
'DEBUG "      ", DEC Dx, " action", CR
BRANCH Dx, [TiltLeft, TiltCenter, TiltRight, StrideLeft,
           StrideCenter, StrideRight]
           ' will fall through if invalid index
RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
  NewValue = LeftTilt
  GOTO MovementTilt

TiltCenter:
  NewValue = CenterTilt
  GOTO MovementTilt

TiltRight:
  NewValue = RightTilt

MovementTilt:
  FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, CurrentStride
    PAUSE MoveDelay
  NEXT

  CurrentTilt = NewValue
  RETURN

StrideLeft:
  NewValue = LeftStride
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  GOTO MovementStride

```

```

StrideRight:
  NewValue = RightStride

MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN

```

### How the Light Compass Works

Program 5.2 takes RC time measurements and first checks to see if the difference between the values returned by the **RCTIME** commands fall in the **DeadBand** variable using the command:

```
IF ABS(LPhotoVal-RPhotoVal) < DeadBand THEN main
```

If the difference between RC times is within the **DeadBand**, the program jumps to the **Main:** label. If the measured difference in RC times is not within the **DeadBand**, two **IF...THEN** statements decide which routine to call, **Turn\_left** or **Turn\_right**.

```
IF ABS(LPhotoVal-RPhotoVal) < DeadBand THEN main

IF LPhotoVal > RPhotoVal THEN turn_right
IF LPhotoVal < RPhotoVal THEN turn_left
```

These routines use the movement routines initially presented in the prior chapter. The Toddler can make smaller turns.

### Your Turn

- √ Enter and run Program Listing 5.2.
- √ Shine a bright flashlight in front of the Toddler. When you move the flashlight, the Toddler should rotate so that it's pointing at the flashlight beam.
- √ Instead of using a flashlight, use your hand to cast a shadow over one of the photoresistors. The Toddler should rotate away from the shadow.
- √ In a darker area, not only will the photoresistor values be larger, so will the difference between them. You may have to increase the deadband in low ambient light to detune the Toddler to small and changing variations in light. The lower the light levels, the less you need the **PAUSE** statements. If the Toddler's performance starts to decrease, it's probably because the time between pulses has exceeded 40 ms. The first line of defense for this problem is to reduce the **PAUSE** period in each subroutine to zero. The second line of defense is to check photoresistors during alternate pulses. That way, after the first pulse, the right photoresistor could be checked. Then, after the second pulse, the left photoresistor could be checked. You can try your hand at developing code that does this in the Challenges section.
- √ Experiment with different ambient light levels and their effect on deadband by trying this experiment in lighter and darker areas. In lighter areas, the deadband value can be made smaller, even zero. In darker areas, the deadband value should be increased.

Swap the conditions in the second and third **IF . . . THEN** statement in Program 5.2. Then re-run the program. Now your Toddler points away from the light.

**ACTIVITY #3: FOLLOWING THE LIGHT**

Programming the Toddler to follow light requires that only a few modifications to Program Listing 5.2 be made. The main change is that measurements within the deadband resulted in no motion in Program Listing 5.2. In Program Listing 5.3, when the difference between RC times falls within the deadband, it results in forward motion. Let's see how it works.

```
' -----[ Title ]-----
' Toddler Program 5.3: Follow the Light
' Marching toward the light
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
TiltServo      CON    13          ' Tilt servo on P12
StrideServo    CON    12          ' Stride servo on P13

LPhotoCircuit  CON    10
RPhotoCircuit  CON    14

' -----[ Constants ]-----
MoveDelay      CON    18          ' in microseconds
TiltStep       CON    5           ' TiltServo step size
StrideStep     CON    5           ' StrideServo step size

RightTilt      CON    620         ' Tilt limits
CenterTilt     CON    750
LeftTilt       CON    880

RightStride    CON    650         ' Stride limits
CenterStride   CON    750
LeftStride     CON    850

DeadBand       CON    5           ' Photoresistor R/C DeadBand

' -----[ Variables ]-----
LPhotoVal      VAR    Word        ' Stores measured R/C times
RPhotoVal      VAR    Word        ' of photoresistors

FigureLoop     VAR    Nib
MoveLoop       VAR    Byte        ' Loop for repeat movements
MoveLoopLimit  VAR    Byte
SubMoveLoop    VAR    Byte        ' Loop for repeat submovements
```

```

SubMoveLoopLmt  VAR   Byte
Pulses          VAR   Word           ' Pulse variable
CurrentTilt     VAR   Word
CurrentStride   VAR   Word
NewValue        VAR   Word

Dx              VAR   Pulses
Mx              VAR   Word
MxCurrent       VAR   Word

Sx              VAR   Word
SxCurrent       VAR   Word

' -----[ EEPROM Data ]-----
'
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL              CON    0
TC              CON    1
TR              CON    2
SL              CON    3
SC              CON    4
SR              CON    5

xx              CON   255

' ----- Movement Value Tables -----
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
'
' Note: ALL movement tables must be in this section

LeftSemicircle DATA   7, bLeftTurn, bLeftTurn, bForward, xx
RightSemicircle DATA  7, bRightTurn, bRightTurn, bForward, xx

WalkForward3     DATA  3, bForward, xx
WalkForward8     DATA  8, bForward, xx

' ----- Basic Movement Codes -----
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish          CON    0
bForward         CON    1
bBackward        CON    2
bLeftTurn        CON    3
bRightTurn       CON    4

```

```

bPivotLeft      CON      5
bPivotRight     CON      6

' ----- Basic Movement Tables -----
'
' These tables can contain Movement Support Codes.

BasicMovements  CON      Forward

Forward         DATA    1, TR, SL, TL, SR, xx
Backward        DATA    1, TR, SL, TL, SR, xx

LeftTurn        DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn       DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft       DATA    1, TL, SR, TC, SL, xx
PivotRight      DATA    1, TR, SL, TC, SR, xx

Finish          DATA    1, TR, SC, TC, xx

' ----- [ EEPROM Data ] -----
GOSUB ResetCC          ' Initialize feet

' ----- [ Main Code ] -----
Main:

' Measure RC time for left photoresistor.
HIGH LPhotoCircuit    ' Set to output-high
PAUSE 3                ' Pause for 3 ms
RCTIME LPhotoCircuit,1,LPhotoVal  ' Measure R/C time on left

' Measure RC time for right photoresistor.
HIGH RPhotoCircuit    ' Set to output-high
PAUSE 3                ' Pause for 3 ms
RCTIME RPhotoCircuit,1,RPhotoVal  ' Measure R/C time on right

' Measure difference between RPhotoVal and LPhotoVal, decide what to do
IF ABS(LPhotoVal-RPhotoVal) > DeadBand THEN check_dir

' Check if difference between RC times is within the deadband
' If yes, then forward.  If no then skip to check_dir subroutine.

walk_forward:
  Mx = Forward
  GOSUB Movement
  goto main

' Select right_turn or left_turn depending on which RC time is larger

```

```

check_dir:
  IF LPhotoVal > RPhotoVal THEN turn_right
  IF LPhotoVal < RPhotoVal THEN turn_left

'----- Navigation Routines -----

turn_left:                                ' turn left towards light
  Mx = PivotLeft
  GOSUB Movement
  goto main                                ' go back to main routine.

Turn_right:                                ' turn right towards light
  Mx = PivotRight
  GOSUB Movement
  GOTO main                                ' go back to main routine.

' -----[ Subroutines ]-----

' ----- Movement: Move feet using DATA table referenced by Mx -----'
'
' Input: Mx = movement table index, table ends in xx
'       or
'       Mx = submovement table index, table ends in xx
'
' Note: All submovement tables come after the movement tables in this file.

Movement:
  IF Mx < BasicMovements THEN SetupMovement
  MxCurrent = Mx                            ' setup to use submovement table
  MoveLoopLimit = 1
  GOTO StartMovement

SetupMovement:
  READ Mx, MoveLoopLimit                    ' read movement table
  MxCurrent = Mx + 1                        ' repeat count

StartMovement:
  FOR MoveLoop = 1 to MoveLoopLimit
    Mx = MxCurrent                          ' Mx = start of movement table
    'debug hex Mx, " Movement ", dec MoveLoop, " of ", dec MoveLoopLimit,cr
    IF Mx < BasicMovements THEN MovementLoop
                                          ' skip if movement table
    SxCurrent = Mx                          ' SxCurrent = submovement index
    GOTO StartSubMovement                  ' enter middle of loop

  MovementLoop:
    READ Mx, SxCurrent                      ' read next submovement byte
    Mx = Mx + 1
    IF SxCurrent = xx THEN MovementDone
                                          ' skip if end of list

```

```

'debug " ", dec SxCurrent, " movement",cr
LOOKUP SxCurrent, [Finish,Forward,Backward,LeftTurn,RightTurn,
PivotLeft,PivotRight], SxCurrent
StartSubMovement:          ' lookup submovement table index
READ SxCurrent, SubMoveLoopLmt ' start executing submovement table
SxCurrent = SxCurrent + 1    ' read submovement table repeat count
FOR SubMoveLoop = 1 to SubMoveLoopLmt
  Sx = SxCurrent

'DEBUG " ", DEC Sx, " submovement ", DEC SubMoveLoop, " of ",
dec SubMoveLoopLmt, CR

  SubMovementLoop:
    READ Sx, Dx              ' read next submovent action
    Sx = Sx + 1
    IF Dx = xx THEN SubMovementDone
                                ' skip if end of list
    GOSUB DoMovement         ' execute movement
    GOTO SubMovementLoop

  SubMovementDone:
NEXT
IF Mx < BasicMovements THEN MovementLoop

  MovementDone:
NEXT
RETURN

DoMovement:
'debug " ", dec Dx, " action",cr
BRANCH Dx, [TiltLeft,TiltCenter,TiltRight,StrideLeft,
StrideCenter,StrideRight]
                                ' will fall through if invalid index
RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
  NewValue = LeftTilt
  GOTO MovementTilt

TiltCenter:
  NewValue = CenterTilt
  GOTO MovementTilt

TiltRight:
  NewValue = RightTilt

MovementTilt:

```



```
FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
  PULSOUT TiltServo, Pulses
  PULSOUT StrideServo, CurrentStride
  PAUSE MoveDelay
NEXT

CurrentTilt = NewValue
RETURN

StrideLeft:
  NewValue = LeftStride
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  GOTO MovementStride

StrideRight:
  NewValue = RightStride

MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
RETURN
```

### How the Light Follower Program Works

As in the previous program, the first **IF...THEN** statement tests for a difference in RC time measurements within the deadband. This statement has been modified so that it skips the **walk\_forward** routine if the difference between RC times falls outside the deadband. On the other hand, if the difference in RC times is within the deadband, the forward pulse is executed. After the forward pulse, the program is directed back to **main** and the RC times are checked again.

```
IF ABS(LPhotoVal-RPhotoVal) > DeadBand THEN check_dir

walk_forward:
  Mx = Forward
  GOSUB Movement
  goto main
```

If the difference between RC times is not within the **DeadBand**, the program skips to the **check\_dir** label. The **IF...THEN** statements following the **check\_dir** label are used to decide whether Toddler should turn left or turn right depending on the inequality of the **RPhotoVal** and **LPhotoVal** values.

```
check_dir:
  IF LPhotoVal > RPhotoVal THEN turn_right
  IF LPhotoVal < RPhotoVal THEN turn_left
```

## CHALLENGES

Lead your Toddler around with a flashlight.

- √ Instead of pointing the photoresistors at the surface directly in front of the Toddler, point them upward and outward. With the photoresistors adjusted this way, the Toddler will roam on the floor and try to always find the brightest place.
- √ Depending on the luminance gradient, you may have to increase the deadband to smooth out the Toddler's light roaming. Alternatively, the deadband may need to be decreased to make it more responsive to seeking out the brighter areas.



## Chapter #6: Object Avoidance with Infrared

### WIRELESS OBJECT DETECTION

Today's hottest products seem to have one thing in common: wireless communication. Personal organizers beam data into desktop computers, and wireless remotes let us channel surf. With a few inexpensive and widely available parts, the BASIC Stamp can also use an infrared LED and detector to detect objects to the front and side of your traveling Toddler.

**Infrared** means Infra means below, so Infra-red is light (or electromagnetic radiation) that has lower frequency, or longer wavelength than red light. Our IR LED and detector work at 980 nm. (nanometers) which is considered near infrared. Night-vision goggles and IR temperature sensing use far infrared wavelengths of 2000-10,000 nm., depending on the application.



Color	Approximate Wavelength (nm)
Violet	400
Blue	470
Green	565
Yellow	590
Orange	630
Red	780
Near infra-red	800-1,000
Infra-red	1,000-2,000
Far infra-red	2,000-10,000

6

Detecting obstacles doesn't require anything as sophisticated as machine vision. A much simpler system will suffice. Some robots use RADAR or SONAR (sometimes called SODAR when used in air instead of water). An even simpler system is to use infrared light to illuminate the robot's path and determine when the light reflects off an object. Thanks to the proliferation of infrared (IR) remote controls, IR illuminators and detectors are easily available and inexpensive.

The Toddler infrared object detection scheme has a variety of uses. The Toddler can use infrared to detect objects without bumping into them. As with the photoresistors, infrared can be used to detect the difference between black and white for line following. Infrared can also be used to determine the distance of an object from the Toddler. The Toddler can use this information to follow objects at a fixed distance, or detect and avoid high ledges.

### **Infrared Headlights for Toddler**

The infrared object detection system we'll build on the Toddler is like a car's headlights in several respects. When the light from a car's headlights reflects off obstacles, your eyes detect the obstacles and your brain processes them and makes your body guide the car accordingly. The Toddler uses infrared LEDs for headlights. They emit infrared, and in most cases, the infrared reflects off objects, and bounces back in the direction of the Toddler. The eyes of the Toddler are the infrared detectors. The infrared detectors send signals to the BASIC Stamp indicating whether or not they detect infrared reflected off an object. The brain of the Toddler, the BASIC Stamp, makes decisions and operates the servo motors based on this input.

The IR detectors have built-in optical filters that allow very little light except the 980 nm infrared that we want to detect onto its internal photodiode sensor. The infrared detector also has an electronic filter that only allows signals around 38.5 kHz to pass through. In other words, the detector is only looking for infrared flashed on and off at 38,500 times per second. This prevents interference from common IR interference sources such as sunlight and indoor lighting. Sunlight is DC interference (0 Hz), and house lighting tends to flash on and off at either 100 or 120 Hz, depending on the main power source in the country where you reside. Since 120 Hz is way outside the electronic filter's 38.5 kHz band pass frequency, it is, for all practical purposes, completely ignored by the IR detectors.

### The FREQOUT Trick

Since the IR detectors only see IR signals in the neighborhood of 38.5 kHz, the IR LEDs have to be flashed on and off at that frequency. A 555 timer can be used for this purpose, but the 555 timer circuit is more complex and less functional than the circuit we will use in this and the next chapter. For example, the method of IR detection introduced here can be used for distance detection; whereas, the 555 timer would need additional hardware to do distance detection.

A pair of Toddler enthusiasts found an interesting trick that made the 555 timer scheme unnecessary. This scheme uses the **FREQOUT** command without the RC filter that's normally used to smooth the signal into a sine-wave. Even though the highest frequency **FREQOUT** is designed to transmit is 32768 Hz, the unfiltered **FREQOUT** output contains a harmonic with useful properties for a 38.5 kHz IR detector. More useful still is the fact that you can use a command such as **FREQOUT Pin, Period, 38500** to send a 38.5 kHz harmonic that the IR detector will detect.

Figure 6-1 shows (a) the signal sent by the command **FREQOUT Pin, Period, 27036**. Tuned electronic receivers, such as the IR detectors we'll be using, can detect components of this signal that are called harmonics. The **FREQOUT** signal's two dominant low frequency harmonics are shown in (b) and (c). (b) shows the fundamental harmonic, and (c) shows the third harmonic. These harmonics are actually components of the unfiltered **FREQOUT** pulses shown in (a). The third harmonic shown in (c) can be controlled directly by entering commands such as **FREQOUT Pin, Period, 38500** (instead of 27036) for 38.5 kHz, or **FREQOUT Pin, Period, 40000** for 40 kHz, etc.

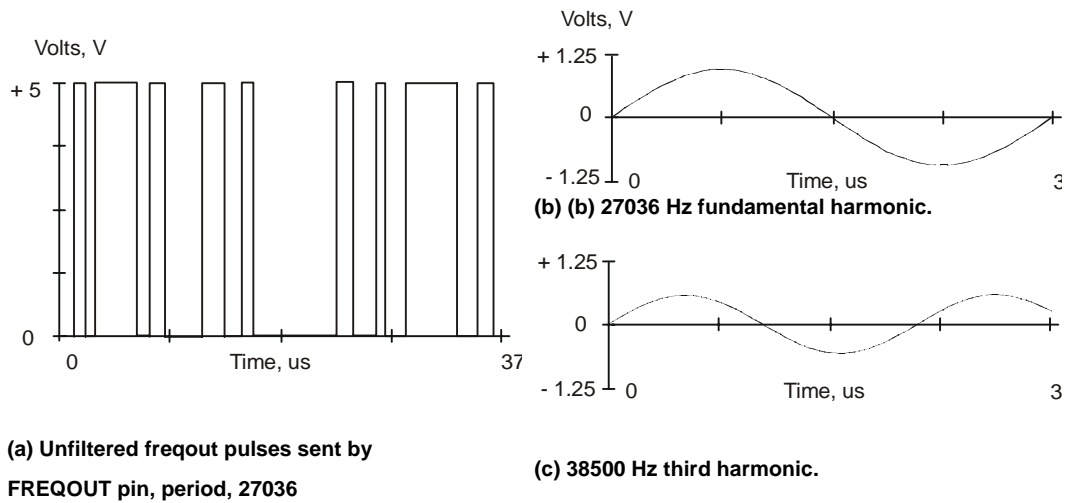


Figure 6-1: FREQOUT Example Properties

Even though the “**FREQOUT**” trick works, there is an additional problem. The BASIC Stamp does not multitask. The reason this is a problem is because the IR detector only sends the low signal indicating that it has detected an object while it is receiving the 38.5 kHz IR. Otherwise, it sends a high signal. Fortunately, it takes the detector long enough to rebound from its low output state that the BASIC Stamp can capture the value. The reason that the detector’s output takes so long to rebound is related to its tendency toward slower responses when it receives a signal with unequal high and low times, of which the signal in (a) has many.



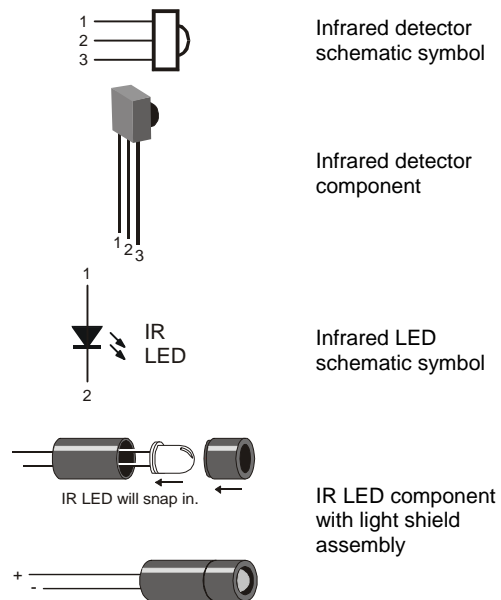
## ACTIVITY #1: BUILDING AND TESTING THE NEW IR TRANSMITTER / DETECTOR

### Parts Required

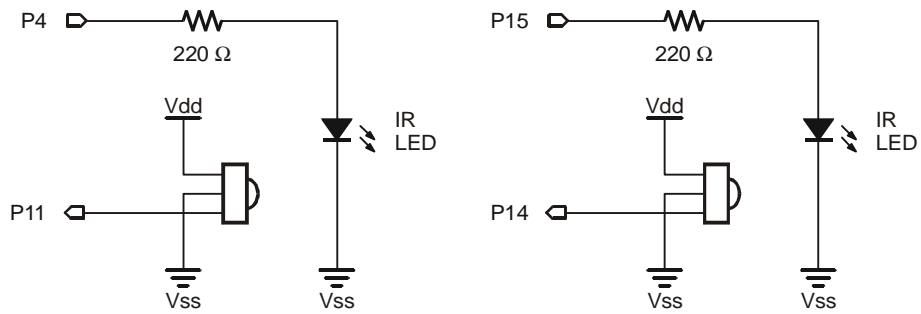
- (1) Piezoelectric speaker
- (2) Shrink wrapped IR LEDs
- (2) IR detectors
- (misc) wires

Figure 6-2 shows the individual part schematics and pictorials so you can identify them. Figure 6-3 is the schematic. Build this circuit on your Toddler board. Note that the 220 ohm resistors are already built into the Toddler PCB; just plug in the infrared LEDs and your Toddler will be ready.

6



**Figure 6-2:**  
Infrared Components and their  
Schematic Symbols



**Figure 6-3:** Infrared circuit schematic Note: the 220 ohm resistors are already built into the Toddler PCB and are not required; simply plug the LEDs into the front sockets.

One IR pair (IR LED and detector) is mounted on each corner of the Toddler circuit board.

### **Testing the IR Pairs**

The key to making each IR pair work is to send 1 ms of unfiltered 38.5 kHz **FREQOUT** harmonic followed immediately by testing the signal sent by the IR detector and saving its output value. The IR detector's normal output state when it sees no IR signal is high. When the IR detector sees the 38500 Hz harmonic sent by the IR LED, its output will drop from high to low. Of course, if the IR does not reflect off an object, the IR detector's output simply stays high. Program 6.1 shows an example of this method of reading the detectors.

```

' -----[ Title ]-----
' Toddler Program 6.1: Infrared Pairs Display
' Test program for the infrared emitters / detectors
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

lEmitter      CON      4
rEmitter      CON      15

lInput        VAR      in11
rInput        VAR      in14

' -----[ Variables ]-----

lDetector      VAR      Bit           ' Two bit variables for saving IR
rDetector      VAR      Bit           ' detector output values.

' -----[ Initialization ]-----

OUTPUT lEmitter           ' signals to function as outputs
OUTPUT rEmitter

' -----[ Main Code ]-----

DO
    ' Detect object on the left.
    ' Send freqout signal - left IR LED
    ' Store IR detector output in RAM.
    ' Detect object on the right.
    ' Repeat for the right IR pair.
    FREQOUT lEmitter, 1, 38500
    lDetector = lInput

    FREQOUT rEmitter, 1, 38500
    rDetector = rInput

    DEBUG home, "Left= ", BIN1 lDetector
    PAUSE 20
    DEBUG "  Right= ", BIN1 rDetector
    PAUSE 20

LOOP

```

### Your Turn

- √ Enter and run Program Listing 6.1.
- √ This program makes use of the BASIC Stamp Editor's Debug Terminal, so leave the serial cable connected to the Toddler while Program Listing 6.1 is running.
- √ While program Listing 6.1 is running, point the IR detectors so nothing nearby could possibly reflect infrared back at the detectors. The best way to do this is to point the Toddler up at the ceiling. The Debug Terminal should display both left and right values as equal to "1."
- √ By placing your hand in front of an IR pair, it should cause the Debug Terminal display for that detector to change from "1" to "0." Removing your hand should cause the output for that detector to return to a "1" state. This should work for each individual detector, and you also should be able to place your hand in front of both detectors and make both their outputs change from "1" to "0."
- √ If the IR Pairs passed all these tests, you're ready to move on; otherwise, check your program and circuit for errors.

### **How the IR Pairs Display Program Works**

Two bit variables are declared to store the value of each IR detector output. The first **FREQOUT** command in the **DO...LOOP** routine is different. The command **FREQOUT lEmitter, 1, 38500** sends the on-off pattern shown in Figure 6-1 via left IR LED circuit by causing it to flash on and off rapidly. The harmonic contained in this signal either bounces off an object, or not. If it bounces off an object and is seen by the IR detector, the IR detector sends a low signal to I/O pin **lInput**. Otherwise, the IR detector sends a high signal to **lInput**. So long as the next command after the **FREQOUT** command is the one testing the state of the IR detector's output, it can be saved as a variable value in RAM. The statement **lDetector = lInput** checks **lInput**, and saves the value ("1" for high or "0" for low) in the **lDetector** bit variable. This process is repeated for the other IR pair, and the IR detector's output is saved in the **rDetector** variable. The **DEBUG** command then displays the values in the Debug Terminal.

### Your Turn

- √ Experiment with detuning your IR pairs by using frequencies above 38.5 kHz. For example, try 39.0, 39.5, 40.0, 40.5 and 41 kHz. Note the maximum distance that each will detect by bringing an object progressively closer to the IR pairs and noting what distance began to cause the IR detector output to switch from “1” to “0.”

### **ACTIVITY #2: OBJECT DETECTION AND AVOIDANCE**

The IR pairs provide range information that the Toddler can use to avoid obstacles. A simple program can simply avoid obstacles providing a random walk around a room without causing a collision. Obstacles must be high enough to be detected by the Toddler’s IR detectors.

**6**

### Sampling Between Movements

Program Listing 6.2 checks the IR pairs and delivers one of four different pulses based on the sensors. Each of the navigational routines is just a single step in the **Forward**, **Left\_turn**, **Right\_turn** or **Backward** directions. After the pulse is applied, the sensors are checked again, then another step is made, etc. This program also makes use of some programming techniques you will find very useful.

```

' -----[ Title ]-----
' Toddler Program 6.2: Object Detection and Avoidance
' Uses the infrared emitters and detectors
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

lEmitter      CON      4
rEmitter      CON      15

lInput        VAR      in11
rInput        VAR      in14

StrideServo   CON      12      ' Stride servo on P12
TiltServo     CON      13      ' Tilt servo on P13

' -----[ Constants ]-----

MoveDelay     CON      18      ' in microseconds
TiltStep      CON      10      ' TiltServo step size
StrideStep    CON      10      ' StrideServo step size

RightTilt     CON      620     ' Tilt limits
CenterTilt    CON      750
LeftTilt      CON      880

RightStride   CON      650     ' Stride limits
CenterStride  CON      750
LeftStride    CON      850

' -----[ Variables ]-----

Sensors       VAR      Nib      ' variable is used to store
                          ' lower two bits of detector values
FigureLoop    VAR      Nib
MoveLoop      VAR      Byte     ' Loop for repeat movements
MoveLoopLimit VAR      Byte

SubMoveLoop   VAR      Byte     ' Loop for repeat submovements
SubMoveLoopLmt VAR     Byte

Pulses        VAR      Word     ' Pulse variable

CurrentTilt   VAR      Word
CurrentStride VAR      Word
NewValue      VAR      Word

Dx            VAR      Pulses
Mx            VAR      Word

```

```

MxCurrent      VAR      Word
Sx             VAR      Word
SxCURRENT      VAR      Word

' -----[ EEPROM Data ]-----
' ----- Movement Support Codes -----
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL             CON      0
TC             CON      1
TR             CON      2

SL             CON      3
SC             CON      4
SR             CON      5

xx            CON      255

' ----- Movement Value Tables -----
'
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
'
' Note: ALL movement tables must be in this section

LeftSemicircle DATA    7, bLeftTurn, bLeftTurn, bForward, xx
RightSemicircle DATA   7, bRightTurn, bRightTurn, bForward, xx

WalkForward3   DATA    3, bForward, xx
WalkForward8   DATA    8, bForward, xx

' ----- Basic Movement Codes -----
'
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish        CON      0
bForward       CON      1
bBackward      CON      2
bLeftTurn      CON      3
bRightTurn     CON      4
bPivotLeft     CON      5
bPivotRight    CON      6

' ----- Basic Movement Tables -----

```

```

'
' These tables can contain Movement Support Codes.

BasicMovements  CON      Forward
Forward          DATA    1, TR, SL, TL, SR, xx
Backward         DATA    1, TR, SR, TL, SL, xx

LeftTurn         DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn        DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft        DATA    3, TL, SR, TC, SL, TR, SR, TC, SL, xx
PivotRight       DATA    3, TR, SL, TC, SR, TL, SL, TC, SR, xx

Finish           DATA    1, TR, SC, TC, xx

' -----[ Initialization ]-----
OUTPUT lEmitter          ' signals to function as outputs
OUTPUT rEmitter

GOSUB ResetCC            ' Initialize feet

' -----[ Main Code ]-----
DO
  FREQOUT lEmitter,1,38500      ' Send freqout signal - left IRLED.
  sensors.bit0 = lInput         ' Store IR detector output in RAM.
                               ' Detect object on the right.
  FREQOUT rEmitter,1,38500      ' Repeat for the right IR pair.
  sensors.bit1 = rInput

  PAUSE 18                    ' 18 ms pause(2 ms lost on freqout)

  ' Loads the IR detector output values into the lower 2 bits of the
  ' sensors variable, a number btwn 0 and 3 that the LOOKUP command can use

  LOOKUP sensors, [Backward,PivotLeft,PivotRight,Forward],Mx

  GOSUB Movement
LOOP

' -----[ Subroutines ]-----
'
' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = movement table index, table ends in xx
'        or
'        Mx = submovement table index, table ends in xx
'
' Note: All submovment tables come after the movment tables in this file.

```



```

Movement:
  IF Mx < BasicMovements THEN SetupMovement

  MxCurrent = Mx                ' setup to use submovement table
  MoveLoopLimit = 1
  GOTO StartMovement

SetupMovement:
  READ Mx, MoveLoopLimit        ' read movement table repeat count
  MxCurrent = Mx + 1

StartMovement:
  FOR MoveLoop = 1 to MoveLoopLimit
    Mx = MxCurrent              ' Mx = start of movement table

'debug DEC Mx, " Movement ", DEC MoveLoop, " of ", DEC MoveLoopLimit,cr

    IF Mx < BasicMovements THEN MovementLoop
      ' skip if movement table
      SxCurrent = Mx            ' SxCurrent = submovement table index
      GOTO StartSubMovement    ' enter middle of loop

      MovementLoop:
        READ Mx, SxCurrent      ' read next submovment byte
        Mx = Mx + 1
        IF SxCurrent = xx THEN MovementDone
          ' skip if end of list
'debug " ", DEC SxCurrent, " movement",cr
        LOOKUP SxCurrent, [Finish,Forward,Backward,LeftTurn,RightTurn,
        PivotLeft,PivotRight], SxCurrent

        StartSubMovement:      ' start executing submovement table
          READ SxCurrent, SubMoveLoopLmt
          ' read submovement table repeat count

          SxCurrent = SxCurrent + 1
          FOR SubMoveLoop = 1 to SubMoveLoopLmt
            Sx = SxCurrent

'debug " ", DEC Sx, " submovement ", DEC SubMoveLoop, " of ",
DEC SubMoveLoopLmt,cr

            SubMovementLoop:
              READ Sx, Dx        ' read next submovent action
              Sx = Sx + 1
              IF Dx = xx THEN SubMovementDone
                ' skip if end of list
              GOSUB DoMovement   ' execute movement
              GOTO SubMovementLoop

            SubMovementDone:

```

```

        NEXT
        IF Mx < BasicMovements THEN MovementLoop
                                ' exit if submovement table
        MovementDone:
        NEXT
        RETURN

    DoMovement:
debug "    ", DEC Dx, " action", cr
        BRANCH Dx, [TiltLeft, TiltCenter, TiltRight, StrideLeft,
        StrideCenter, StrideRight]
                                ' will fall through if invalid index
        RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
    NewValue = LeftTilt
    GOTO MovementTilt

TiltCenter:
    NewValue = CenterTilt
    GOTO MovementTilt

TiltRight:
    NewValue = RightTilt

MovementTilt:
    FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
        PULSOUT TiltServo, Pulses
        PULSOUT StrideServo, CurrentStride
        PAUSE MoveDelay
    NEXT

    CurrentTilt = NewValue
    RETURN

StrideLeft:
    NewValue = LeftStride
    GOTO MovementStride

StrideCenter:
    NewValue = CenterStride
    GOTO MovementStride

StrideRight:
    NewValue = RightStride

MovementStride:
    FOR Pulses = CurrentStride TO NewValue STEP StrideStep
        PULSOUT TiltServo, CurrentTilt

```

```

    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
NEXT

CurrentStride = NewValue
RETURN

' ----- Move feet to initial center position -----

ResetCC:
CurrentTilt    = CenterTilt
CurrentStride  = CenterStride

FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
NEXT

DoReturn:
RETURN

```

### How IR Roaming by Numbers in Real-Time Works

This Program listing declares the **sensors** variable, which is one nibble of RAM. Of the four bits in the sensors variable, only the lowest two bits are used. Bit-0 is used to store the left detector's output, and bit-1 is used to store the right detector's output.

The main routine starts with the **FREQOUT** commands used to send the IR signals, but the commands following each freqout command are slightly different from those used in the previous program. Instead of saving the bit value at the input pin to a bit variable, each bit value is stored as a bit in the **sensors** variable. Bit-0 of **sensors** is set to the binary value of **IN8**, and bit-1 of the **sensors** variable is set to the binary value of **IN0**. After setting the values of the lower two bits of the **sensors** variable, it will have a decimal value between "0" and "3." The **BRANCH** command uses these numbers to determine to which label it sends the program.

```

DO
    FREQOUT lEmitter,1,38500      ' Send freqout signal - left IRLED.
    sensors.bit0 = lInput        ' Store IR detector output in RAM.
                                ' Detect object on the right.
    FREQOUT rEmitter,1,38500      ' Repeat for the right IR pair.
    sensors.bit1 = rInput

```

```

PAUSE 18                                ' 18 ms pause(2 ms lost on freqout)

' Loads the IR detector output values into the lower 2 bits of the
' sensors variable, a number btwn 0 and 3 that LOOKUP can use

LOOKUP sensors, [Backward,PivotLeft,PivotRight,Forward],Mx

GOSUB Movement
LOOP
    
```

The four possible binary numbers that result are shown in Table 6.1. Also shown is the lookup action that occurs based on the value of the state argument.

Binary Value of state	Decimal Value of State	What the Value Indicates, Branch Action Based on State
0000	0	lInput = 0 and rInput = 0, Both IR detectors detect object, step back.
0001	1	lInput = 0 and rInput = 1, Left IR detector detects object, turn right.
0010	2	lInput = 1 and rInput = 0, Right IR detector detects object, turn left.
0011	3	lInput = 1 and rInput = 1, Neither IR detector detects object, step forward.

**Table 6-1:** IR Detector States as Binary Numbers

The **Mx** variable is set to the appropriate movement table index. The **Movement** routine then performs the appropriate sequence of commands.

## CHALLENGES

You can rearrange the address labels in the `LOOKUP` command so that the Toddler does different things in response to obstacles. One interesting activity is to try replacing the `Backward` address with the `Forward` address. There will be two instances of `Forward` in the `Lookup` address list, but this is not a problem. Also, swap the `Left_turn` and `Right_turn` addresses.

√ Try making the changes just discussed.

The Toddler should now run into your hand. Because of this, one Toddler cannot be programmed to follow another without some way of distance detection. If the one in front stops, the one in back will crash into it. This problem will be fixed as an example in the next chapter.



## Chapter #7: Staying on the Table

---

### WHAT'S A FREQUENCY SWEEP?

In general, a frequency sweep is what you do when checking your favorite radio stations. Set the station for one frequency, and check the output. If you don't like the song that's playing, change the frequency and check the output again.

### ACTIVITY #1: TESTING THE FREQUENCY SWEEP

The Toddler can be programmed to send different IR frequencies, and to check for object detection at each frequency. By keeping track of the frequencies for which the IR detector reported an object, its distance can be determined. The left axis of the graph in Figure 7.1 shows how the sensitivity of the IR detector's electronic filter decreases as it receives frequencies greater than 38.5 kHz. The filter essentially causes the IR detector to become less able to detect IR at these frequencies. Another way to think about it is that you have to move an object closer if you want it to be detected at a less sensitive frequency. Since the detector is less sensitive, it will take brighter IR (or a closer object) to make the detector see the signal.

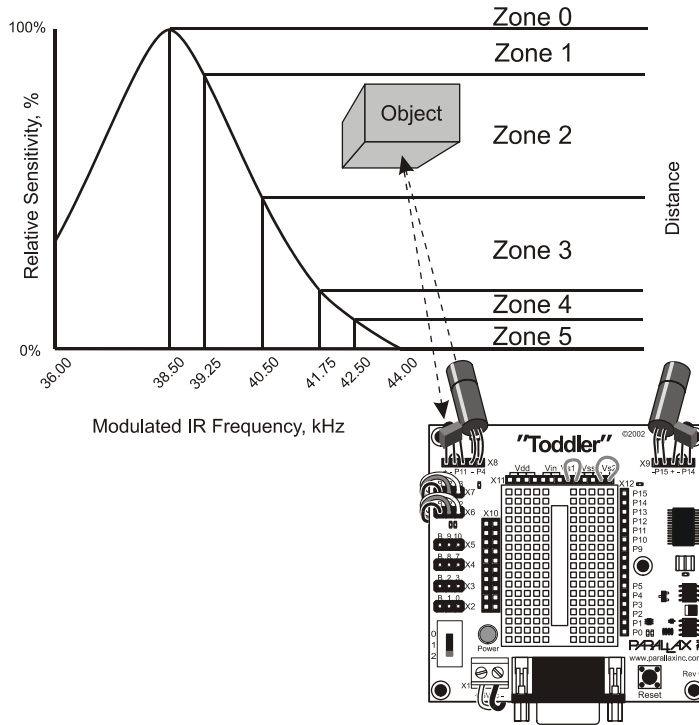


Figure 7-1: Relative IR Sensitivity to Frequency

Figure 7-1 compares the left axis of the graph (IR frequency) to the relative sensitivity of the IR detector. The right side of the graph shows how the relative sensitivity of the IR detector relates to distance detection. As detector sensitivity decreases with the increase in frequency, the object must be closer for the IR signal to be detected. Why closer? When the detectors are made less sensitive by sending higher frequencies, it's like giving them darker and darker lenses to look through. Just as a flashlight beam appears brighter when reflected off an object that's closer to you, IR reflected off a closer object appears brighter to the IR detectors.

The right axis of Figure 7-1 shows how different frequencies can be used to indicate in which zone a detected object is located. By starting with a frequency of 38.5 kHz, whether or not an object is in Zone 1-5 can be determined. If an object is not yet detected, it must be beyond the detector limit (Zone 0). If an object is detected, by testing again at 39.25 kHz, the first datum about distance is collected. If 38.5 kHz is detected the



object but 39.25 kHz did not, the object must be in Zone 1. If the object was detected at both frequencies, but not at 40.5 kHz, we know it's in Zone 2. If all three frequencies detected the object, but it was not detected at 41.75 kHz, we know it is in Zone 3. If all four frequencies detected the object, but not 42.5 kHz, we know it's in Zone 4. If all the frequencies detected the object, we know it's in Zone 5.

The frequency sweep technique used in this chapter works fairly well for the Toddler, and the components are only a fraction of the cost of common IR distance sensors. The trade off is that the accuracy of this method is also only a fraction of the accuracy of common IR distance sensors. For basic Toddler tasks that require some distance perception, such as following another Toddler, this interesting technique does the trick. Along with adding low-resolution distance perception to the Toddler's senses, it also provides an introduction to the concepts of filters and frequency response.

### **Parts Required**

Use the same IR detection circuit from Chapter 6, shown in Figure 6.4, for this activity.

**7**

### **Programming the IR Distance Gauge**

Programming the BASIC Stamp to send different frequencies involves a `DO...LOOP`. The `Counter` variable can be used to give the `FREQOUT` command different frequencies to check. This program introduces the use of arrays. Arrays are used in Program 7.1 to store the IR detector outputs at the different frequencies. For the `L_values` variable, the Zone 0 output is stored in bit-0 of `L_values`. The Zone 1 output is stored in bit-1 `L_values.bit1`, and so on, all the way through Zone 5, which is stored in bit-5 of `L_values`. The same measurements are taken for `R_values`.

```

' -----[ Title ]-----
' Toddler Program 7.1: IR Distance Gauge
' Test of infrared sensors to show distance measurement
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
LeftIRLED      CON      4
RightIRLED     CON      15
LeftDetector   VAR      IN11
RightDetector  VAR      IN14

' -----[ Variables ]-----
Counter        VAR      Nib           ' Counting variable
L_values       VAR      Byte          ' Vars for storing freq
R_values       VAR      Byte          ' sweep IR detector outputs
IR_freq       VAR      Word           ' Frequency argument

' -----[ Main Routine ]-----
DO
  L_values = 0                          ' Reset L_values and
  R_values = 0                          ' R_values to 0

  ' Load sensor outputs into L_values and R_values using a FOR..NEXT loop
  ' and a lookup table, and bit addressing

  FOR Counter = 0 TO 4

    LOOKUP counter, [37500,38250,39500,40500,41500], IR_freq

    FREQOUT LeftIRLED,1, IR_freq
    L_values.lowbit(counter) = ~LeftDetector

    FREQOUT RightIRLED,1, IR_freq
    R_values.lowbit(counter) = ~RightDetector

  NEXT

  ' Display l_values and r_values in binary and ncd format.

  DEBUG HOME, CR, CR, "Left readings      Right Readings", cr
  DEBUG " ",BIN8 L_values, "          ", BIN8 R_values, cr
  DEBUG " ",DEC5 NCD(L_values), "          ", DEC5 NCD(R_values), CR

LOOP

```

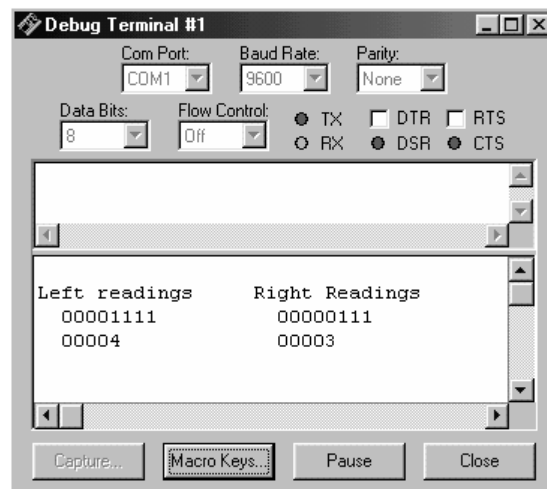
**Your Turn**

- √ Enter and run Program Listing 7.1.

This program makes use of the Debug Terminal, so leave the serial cable connected to the Toddler while Program Listing 7.1 is running.

When the Toddler is placed facing a nearby wall (3 to 5 cm.), the Debug Terminal should display something similar to Figure 7.2. As the Toddler is moved closer to and further from the wall, the numbers displayed by the Debug Terminal should change increase and decrease. Each “1” represents a zone so that when you see five 1’s the object is nearest to the Toddler.

- √ Place the Toddler so that it faces the wall with its IR LEDs about 1 cm. away from the wall. The left and right readings should both be at “4” or “5.” If not, make sure each IR detector is facing in the same direction as its IR LED.



**Figure 7-2:** Frequency sweep and data in binary and NCD format

- √ Gradually back the Toddler away from the wall. As the Toddler is backed away from the wall, the left and right readings should gradually decrease to “0.”

- √ If either or both sides stay at all zeros or all ones, it indicates a possible mistake in either your wiring or in the program. If this is the case, unplug your battery pack from the Toddler. Then, check your wiring and PBASIC code for errors.

The maximum detection distance is 20 to 30 cm., depending on the reflectivity of the wall. Some tinkering with how far left/right each IR pair is pointing may be required to get the numbers to be the same at a given distance. A high level of precision IS NOT necessary for these activities.



**IR circuit not working?** Use a wire stripper to unsheathe about 1 cm of insulation from a jumper wire. Slide the insulation up one of the IR LED leads. This will protect the leads from touching each other during adjustment.

### How the Distance Gauge Program Works

**Counter** is a nibble variable that is used to index a **FOR...NEXT** loop. The **FOR...NEXT** loop is used for checking the IR detectors at various frequencies. The **L\_values** and **R\_values** variables store the outputs for the left and right IR detectors at the various frequencies used. Each variable stores five binary measurements. Since the IR detector outputs are tested at a variety of frequencies, **IR\_freq** is a variable that can store the value of the frequency that gets sent each time through the frequency testing loop.

The main routine contains two routines, one for frequency sweep and another for displaying the data collected. The first step in the frequency sweep is setting **L\_values** and **R\_values** to zero. This is important since individual bits in each variable are modified. Clearing **L\_values** and **R\_values** starts each variable with a clean slate. Then individual bits can be set to “1” or “0,” depending on what the IR detectors report.

```
DO
    L_values = 0
    R_values = 0
```

The **FOR. . . NEXT** loop is where the frequency sweep occurs. The lookup command checks the counter value to determine which frequency to copy to the **IR\_freq** variable. When counter is “0,” 37500 gets copied to **IR\_freq**. When counter is “1,” 38250 is copied to **IR\_freq**. As the value of counter is incremented from “0” to “4” by the **FOR. . . NEXT** loop, each successive value in the lookup table is copied to **IR\_freq**.

```
FOR Counter = 0 to 4
```

```
LOOKUP counter, [37500,38250,39500,40500,41500], IR_freq
```

Note that the lookup table begins the frequency sweep at 37500 (most sensitive) and ends at 41500 (least sensitive). You might be wondering why the numbers in the `LOOKUP` table don't match the frequency values from Figure 7.1. It's true that if the BASIC Stamp could transmit a 50% duty cycle pulse train (pulses with the same high time and low time) at these frequencies, they would have to match the frequencies specified for the IR detector's filter. However, the `FREQOUT` command introduces other factors that affect the amplitude of the harmonics transmitted by the IR LEDs. The math involved in predicting the optimum frequency arguments to use is very advanced and is well outside the scope of this text. Even so, the best frequencies for a given distance can be determined experimentally. The list of values we are using are known to be reliable.

The left sensor is checked by using `FREQOUT` to send the current value of `IR_freq`. Next, the `.lowbit()` argument is used to address each successive bit in `L_values`. When counter is "0," the `.lowbit(counter)` argument addresses bit-0 of `L_values`. When counter is "1," the `.lowbit(counter)` argument addresses bit-1 of `L_values`, and so on. Before writing the value of `IN8` to `L_values.lowbit(counter)`, the `NOT` operator (`~`) is used to invert the bit's value before it is stored to its bit array location in `L_values`. The same process is then repeated for `R_values`. After the fifth time through the `FOR...NEXT` loop, the IR data bits have all been loaded into `L_values` and `R_values`.

```
FREQOUT LeftIRLED,1, IR_freq
L_values.lowbit(counter) = ~LeftDetector

FREQOUT RightIRLED,1, IR_freq
R_values.lowbit(counter) = ~RightDetector

NEXT
```

The `DEBUG` commands use a variety of formatters and text strings to display the `L_values` and `R_values` variables. The first row of the display is the text heading indicating which readings correspond the right IR detector and which readings correspond to the left IR detector. Remember that left and right are treated as though you are sitting in the Toddler's body.

```
DEBUG HOME, CR, CR, "Left readings      Right Readings", cr
DEBUG " ",BIN8 L_values, "          ", BIN8 R_values, cr
DEBUG " ",DEC5 NCD(L_values), "      ", DEC5 NCD(R_values),
CR
```

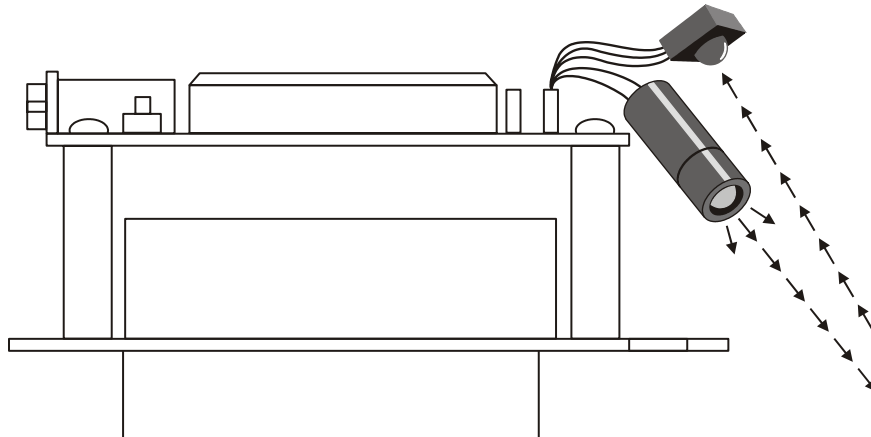
The second row displays `L_values` and `R_values` in binary format. This allows for observation of how the bit values in `L_values` and `R_values` change as the apparent distance of an object changes.

The third row displays the `NCD` value of each variable. The `NCD` operator returns a value that corresponds to the location of the most significant bit in a variable. If the variable is all zeros, `NCD` returns a zero. If the least significant bit contains a “1,” and all the rest of the digits are “0,” `NCD` returns a “1.” If bit-1 contains a “1,” but all the numbers to the left of bit-1 are zeros, `NCD` returns a “2,” and so on. The `NCD` operator is a handy way of indicating how many ones have been loaded into the lower bits of `L_values` and `R_values`. What’s really handy is that `NCD` directly tells you in which zone the object has been detected.

When the display routine is finished sending data to the Debug Terminal, program control is returned to the `main` label.

### **Your Turn**

- √ With Program 7.1 running, place the Toddler facing the wall so that the IR LEDs are about 1.5 cm. from the wall. For best results, tape a white sheet of paper to the wall.
- √ Make a note of the left and right readings.
- √ Start pulling the Toddler away from the wall.
- √ Each time the value of one or the other sensors decreases, make a note of the distance. In this way you can determine the zones for each of your Toddler’s IR pairs.
- √ If the readings on one side are consistently larger than the other, you can point the IR LED on the side reporting the larger readings outward a little further. For example, if the left IR pair continually reports higher readings than the right IR pair, try pointing the left IR LED and detector a little further to the left.

**ACTIVITY #2: THE DROP-OFF DETECTOR**

7

**Figure 7-3: IR LED and Receiver Adjustment for Edge Detection**

One application for distance detection is checking for a drop-off. For example, if the Toddler is navigating on a table, it can change direction if it sees the edge of the table. All you have to do is point the IR pairs downward so that they are both pointing at the table right in front of the Toddler. A distance detection program can then be used to detect that the table is close-up. When the Toddler nears the edge of a table, one or both of the distance detectors will start reporting that they no longer see something close-up. That means it's time to turn away from the abyss. This program works best on a light-colored table. Darker tables will absorb more light and be less useful at reflecting infrared.

- √ Point your IR pairs at the surface directly in front of the Toddler as shown in Figure 7.3. The IR pairs should be pointed downward at least  $45^\circ$  from horizontal and outward  $45^\circ$  from the Toddler's center line.
- √ Perform the tests below using Program 7.1 before trying Program 7.2.
- √ Record the IR pair outputs when the Toddler is looking straight at the table. If the values of the IR pairs when they are looking at your tabletop are "3" or more, it indicates your detectors are seeing what they are supposed to see.

- √ Record the IR pair outputs when the Toddler is looking off the edge of the table. If these values remain less than “3,” the Toddler is ready to try Program Listing 7.2.
- √ If the Toddler does not give you steady and consistent readings of “3” or more when the Toddler is looking at the table, try first adjusting the direction the IR pairs are pointing. Also, if the Toddler does not consistently register less than “3” when it’s looking off the edge of the table, some additional adjustment of the IR pairs also is in order.
- √ If the sensors report “3” or more while looking at the table and “2” or less when looking off the edge, the Toddler is ready for Program Listing 7.2.

Make sure to be the spotter for your Toddler when running Program Listing 7.2. Always be ready to pick your Toddler up as it approaches the edge of the table it’s navigating. If the Toddler tries to drive off the edge, pick it up before it takes the plunge. Otherwise, your Toddler might become a Not-Bot!

When spotting your Toddler while it’s avoiding drop-offs, be ready to pick it up from above. Otherwise, the Toddler will see your hands instead of the drop-off and not perform as expected..

Program Listing 7.2 uses modified versions of the **forward**, **right\_turn**, **left\_turn** and **backward** routines that have been used and reused in every chapter since Chapter #2. The number of pulses in each routine has been adjusted for better performance along a table edge. The **check\_sensors** subroutine takes distance measurements by recycling code from Program Listing 7.1: IR Distance Gage.

Run and test Program Listing 7.2. Remember, always be ready to pick your Toddler up if it tries to run off the table.

---



```

' -----[ Title ]-----
' Toddler Program 7.2: Drop-off Detection
' Walking on a table avoiding the edges
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
TiltServo      CON      13          ' Tilt servo on P13
StrideServo    CON      12          ' Stride servo on P12

' -----[ Constants ]-----
MoveDelay      CON      25          ' in microseconds
TiltStep       CON      10          ' TiltServo step size

StrideStep     CON      10          ' StrideServo step size
RightTilt      CON      620         ' Tilt limits
CenterTilt     CON      750
LeftTilt       CON      880

RightStride    CON      650         ' Stride limits
CenterStride   CON      750
LeftStride     CON      850

' -----[ Variables ]-----
FigureLoop     VAR      Nib
MoveLoop       VAR      Byte       ' Loop for repeat movements
MoveLoopLimit  VAR      Byte

SubMoveLoop    VAR      Byte       ' Loop for repeat submovements
SubMoveLoopLmt VAR      Byte

Pulses         VAR      Word       ' Pulse variable

CurrentTilt    VAR      Word
CurrentStride  VAR      Word
NewValue      VAR      Word

Dx             VAR      Pulses

Mx             VAR      Word
MxCurrent     VAR      Word

Sx            VAR      Word
SxCurrent     VAR      Word

```

```
' -----[ EEPROM Data ]-----
'
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL          CON      0
TC          CON      1
TR          CON      2

SL          CON      3
SC          CON      4
SR          CON      5

xx          CON      255

' ----- Movement Value Tables -----
'
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
'
' Note: ALL movement tables must be in this section

LeftSemicircle DATA  7, bLeftTurn, bLeftTurn, bForward, xx
RightSemicircle DATA  7, bRightTurn, bRightTurn, bForward, xx

WalkForward3   DATA  3, bForward, xx
WalkForward8   DATA  8, bForward, xx

' ----- Basic Movement Codes -----
'
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish       CON      0
bForward      CON      1
bBackward     CON      2
bLeftTurn     CON      3
bRightTurn    CON      4
bPivotLeft    CON      5
bPivotRight   CON      6

' ----- Basic Movement Tables -----
'
' These tables can contain Movement Support Codes.

BasicMovements CON      Forward
```

```

Forward      DATA    1, TR, SL, TL, SR, xx
Backward     DATA    1, TR, SR, TL, SL, xx

LeftTurn     DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn    DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft    DATA    3, TL, SR, TC, SL, TR, SR, TC, SL, xx
PivotRight   DATA    3, TR, SL, TC, SR, TL, SL, TC, SR, xx

Finish       DATA    1, TR, SC, TC, xx

'----- Local Declarations -----

counter      VAR      Nib          ' For...next loop index variable
l_values     VAR      Mx           ' R sensor vals for processing
r_values     VAR      Sx           ' L sensor vals for processing
l_IR_freq    VAR      MxCurrent    ' L IR freqs from lookup table
r_IR_freq    VAR      SxCurrent    ' R IR freqs from lookup table

lEmitter     CON      4
rEmitter     CON      15

lDetector    VAR      IN11
rDetector    VAR      IN14

' -----[ Initialization ]-----

OUTPUT lEmitter          ' Set infrared emitters to outputs
OUTPUT rEmitter
OUTPUT 2
FREQOUT 2,500,3000       ' Signal program start
GOSUB ResetCC

' -----[ Main Routine ]-----

Main:                  ' Main routine

' The command "gosub check_sensors" sends the program to a subroutine
' that loads distance values into l_values and r_values. So, when the
' fprogram returns rom the check_sensors subroutine, the values are
' updated and ready for distance based decisions.

GOSUB check_sensors

' The distances are checked for four different inequalities. Depending
' on the inequality that turns out to be true, the program either
' branches to the forward, left_turn, right_turn or backward navigation
' routine. The "3" value used below to test the boundary conditions
' may need to be changed depending upon the color of the walking surface
' and the angle of IR LEDs and detectors.

```

```

Boundary      CON      2

IF l_values >= boundary AND r_values >= boundary THEN go_forward
IF l_values >= boundary AND r_values < boundary THEN left_turn
IF l_values < boundary AND r_values >= boundary THEN right_turn
IF l_values < boundary AND r_values < boundary THEN go_backward

GOTO main          ' Repeat the process.

'----- Navigation Routines -----

go_forward:        ' single forward pulse, then
  Mx = Forward
  GOSUB Movement
  GOTO main        ' go back to the main: label.

left_turn:         ' eight left pulses, then
  Mx = PivotLeft
  GOSUB Movement
  GOTO main        ' go back to the main: label.

right_turn:        ' eight right pulses, then
  Mx = PivotRight
  GOSUB Movement
  GOTO main        ' go back to the main: label.

go_backward:       ' eight backward pulses, then
  Mx = Backward
  GOSUB Movement
  GOTO main        ' go back to the main: label.

' -----[ Subroutines ]-----

' The check sensors subroutine is a modified version of Program Listing
' 6.1 without the debug Terminal display. Instead of displaying l_values
' and r_values, the main routine uses these values to decide which way to
' go.

check_sensors:

  l_values = 0      ' Reset l_values and r_values to 0.
  r_values = 0

  ' Load sensor outputs into l_values and r_values using a FOR..NEXT loop
  ' a lookup table, and bit addressing.

  FOR counter = 0 TO 4

    check_left_sensors:
      LOOKUP counter, [37500,38250,39500,40500,41500], l_IR_freq

```

```

        FREQOUT lEmitter, 1, l_IR_freq
        l_values.lowbit(counter) = ~ lDetector

    check_right_sensors:
        LOOKUP counter, [37500,38250,39500,40500,41500], r_IR_freq
        FREQOUT rEmitter, 1, r_IR_freq
        r_values.lowbit(counter) = ~ rDetector
    NEXT

' Convert l_values and r_values from binary to ncd format.

    l_values = ncd l_values
    r_values = ncd r_values

' Now l_values and r_values each store a number between 0 and 5
' corresponding to the zone the object is detected in. The program can
' now return to the part of the main routine that makes decisions based
' on these distance measurements.

RETURN

' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = movement table index, table ends in xx
'       or
'       Mx = submovement table index, table ends in xx
'
' Note: All submovement tables come after the movement tables in this file.

Movement:
    IF Mx < BasicMovements THEN SetupMovement
    MxCurrent = Mx                ' setup to use submovement table
    MoveLoopLimit = 1
    GOTO StartMovement

SetupMovement:
    READ Mx, MoveLoopLimit        ' read movement table repeat count
    MxCurrent = Mx + 1

StartMovement:
    FOR MoveLoop = 1 to MoveLoopLimit
        Mx = MxCurrent            ' Mx = start of movement table
        DEBUG DEC Mx, " Movement ", dec MoveLoop, " of ", dec MoveLoopLimit, CR

        IF Mx < BasicMovements THEN MovementLoop
            ' skip if movement table

        SxCurrent = Mx           ' SxCurrent = submovement index
        GOTO StartSubMovement    ' enter middle of loop

    MovementLoop:
        READ Mx, SxCurrent        ' read next submovement byte

```

```

Mx = Mx + 1
IF SxCurrent = xx THEN MovementDone
                                ' skip if end of list
DEBUG " ", DEC SxCurrent, " movement", CR
LOOKUP SxCurrent, [Finish, Forward, Backward, LeftTurn, RightTurn,
PivotLeft, PivotRight], SxCurrent
StartSubMovement:                ' lookup submovement table index
READ SxCurrent, SubMoveLoopLmt    ' start executing submovement table
                                ' read submovement table repeat
SxCurrent = SxCurrent + 1

FOR SubMoveLoop = 1 TO SubMoveLoopLmt
  Sx = SxCurrent

DEBUG " ", DEC Sx, " submovement ", DEC SubMoveLoop, " of "
DEBUG DEC SubMoveLoopLmt, CR

  SubMovementLoop:
  READ Sx, Dx                    ' read next submovent action
  Sx = Sx + 1

  IF Dx = xx THEN SubMovementDone ' skip if end of list
  GOSUB DoMovement                ' execute movement
  GOTO SubMovementLoop

  SubMovementDone:
  NEXT
  IF Mx < BasicMovements THEN MovementLoop
                                ' exit if submovement table
  MovementDone:
  NEXT
RETURN

DoMovement:
DEBUG " ", dec Dx, " action", cr
BRANCH Dx, [TiltLeft, TiltCenter, TiltRight, StrideLeft, StrideCenter,
StrideRight]
                                ' will fall through if invalid
RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
  NewValue = LeftTilt
  GOTO MovementTilt

TiltCenter:
  NewValue = CenterTilt

```

```
GOTO MovementTilt

TiltRight:
  NewValue = RightTilt

MovementTilt:
  FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, CurrentStride
    PAUSE MoveDelay
  NEXT

  CurrentTilt = NewValue
  RETURN

StrideLeft:
  NewValue = LeftStride
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  GOTO MovementStride

StrideRight:
  NewValue = RightStride

MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN
```

### Aliased Variables

The Drop-off Detection program in Program 7.2 is the beginning of a rather large program in terms of data memory. In fact, without a little PBASIC programming trick, the program will not compile. The trick is PBASIC's ability to alias a variable so it uses the storage space of another variable. This allows the program to run with the 16 words of RAM space (actually 3 words are used for the BASIC Stamp's PBASIC and interface pin support).

The following code from Program Listing 7.2 shows how the aliasing is done.

```

counter      var    nib
l_values     var    Mx
r_values     var    Sx
l_IR_freq   var    MxCurrent
r_IR_freq   var    SxCurrent

```

The first **VAR** definition is normal. It defines a nibble variable. The next four reuse different variables. They are the same size as the aliased variables. The main requirement to keep in mind when using aliased variables is that any variables sharing the same storage that these variables cannot be used at the same time. In other words, do not try the following.

```

l_values = 1
Mx = 2

```

Aliasing is normally used because the original variable names do not work well with a new part of the program or subroutine. PBASIC has no concept of local variables so aliasing is required.



The BASIC Stamp's IDE can present the memory map of the current program. This provides RAM and EEPROM usage information. The memory map for the Toddler Program 7.2 is shown in Figure 7.4. It shows 5 bytes of free RAM. Not much but enough. This includes the use of four word aliased variables. If these variables were not aliased then the program would need additional 8 bytes, 3 more than available.

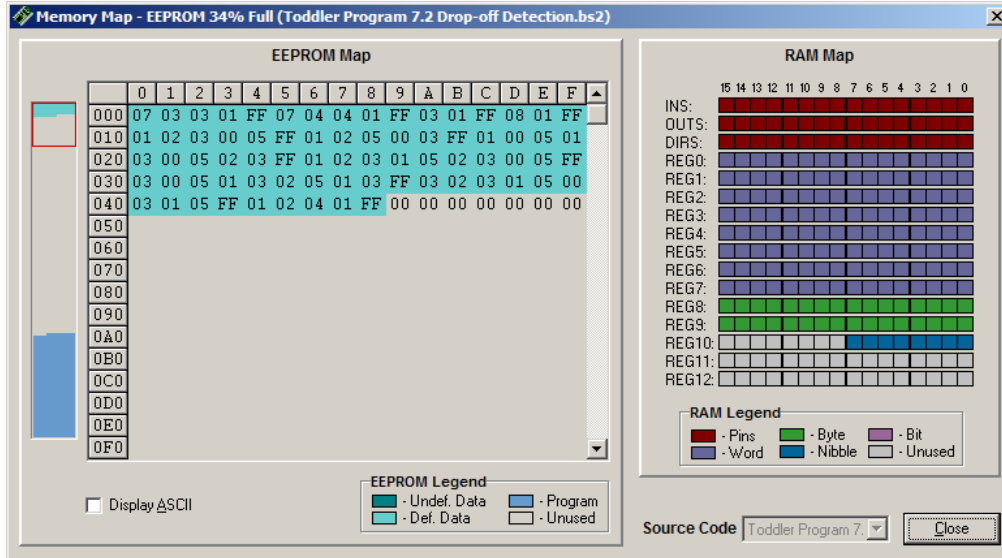


Figure 7-4: EEPROM Memory Map for Toddler Program 7.2

Aliasing should be used with great care. It is a significant source of problems when debugging a program. The advantage of using this with the BASIC Stamp is that only a limited number of variables will be used in the program so it is readily apparent where problems occur.

In this case, the initial set of variables including **Mx** is used in the movement part of the program. Only the **Mx** variable is used outside of the **Movement** routine and that is used to pass a parameter to the routine. The aliased variables including **l\_IR\_freq** variable is used in the range finding routine. Since these two routines do not call each other it is easy to isolate the two with respect to variables.

### How the Drop-off Avoidance Program Works

Now that we have the aliasing issue out of the way we can move onto the main program. The first thing the main routine does is call the `check_sensors` subroutine. Note that `check_sensors` is simply Program 7.1 with no Debug Terminal display placed in a subroutine. Instead of debugging the NCD values of `l_detect` and `r_detect`, the values of these two variables are simply converted to NCD values using the statements:

```

        l_values = ncd l_values
and
        r_values = ncd r_values

```

After calling the `check_sensors` subroutine, `l_values` and `r_values` are numbers between “0” and “5.” After the program returns from the `check_sensors` subroutine, `l_values` and `r_values` are checked against the benchmarks distance indicating the edge of the table has been detected.

```
boundary CON 2
```

```

IF l_values >= boundary AND r_values >= boundary THEN go_forward
IF l_values >= boundary AND r_values < boundary THEN left_turn
IF l_values < boundary AND r_values >= boundary THEN right_turn
IF l_values < boundary AND r_values < boundary THEN go_backward

```

The routines then load the `Mx` variable with the index of the appropriate table. The `Movement` routine then uses the table to initiate the Toddler’s leg movements. The boundary value is the distance boundary condition. This may need to be changed depending upon the color of the surface the Toddler is walking on. It must be set so that the Toddler reliably sees the table when moving forward.

The angle at which the IR LEDs and sensors can be tilted downward is limited so a low boundary value is typical. One alternative to having a value of 1 or 2 is to adjust the range finding frequencies so that the midrange values are sensing distances farther away. The other alternative is to mount the IR LEDs and sensors closer or on to the Toddler’s feet.

The current configuration with the IR LEDs and sensors mounted on the Toddler’s circuit board does lead to a long range recognition of the edge of the table so the Toddler should not get much closer than a foot from the edge. This means the Toddler needs a relatively large table with a white or light colored surface to walk on.

The Toddler will also attempt to walk along the edge of the table although this is not explicitly built into the program. In theory, if the Toddler walks perpendicular to the edge it will walk to the edge, back up, walk forward and repeat this indefinitely. In practice, this does not occur for two reasons. The first is that the Toddler's movement is not perfectly repeatable. As it moves forward and backwards, the Toddler turns slightly to the one side or the other. Eventually the IR sensors will detect the difference and the Toddler will turn instead of backing up or moving forward.

The sensors themselves are another area that will cause the Toddler to turn parallel to the edge of the table. This will occur if one sensor is more sensitive than the other. Of course, this difference will work in one direction and may cause the Toddler to take an extra step forward if the detection is handled by the other side. This will not cause the Toddler to walk off the table though since it tries to stay so far away from the edge. An extra step or two will not cause a problem.

One area that can be a problem especially when the IR LEDs and sensors are pointed forward is that the Toddler will have limited peripheral vision. It is possible for the Toddler to turn parallel to an edge and drift towards the edge. In theory, the sensor on that side should detect the edge and the Toddler will turn away from the edge. This problem occurs more often when the edge of the table is irregular. Aiming the IR LEDs and sensors outward slightly can help eliminate the problem if it occurs.

7

### **ACTIVITY #3: TODDLER SHADOW WALKER**

For one Toddler to follow another, the Toddler that follows, a.k.a. the shadow walker, has to know how far the lead Toddler is ahead. If the shadow Toddler is lagging behind, it has to detect this and speed up. If the shadow Toddler is too close to the lead Toddler, it has to detect this as well and slow down. If it's the right distance, it can wait until the measurements indicate it's too far or too close again.

Unlike the Toddler's sibling the Boe-Bot, the Toddler moves in discrete steps, not small increments using wheels. Whereas the Boe-Bot uses calculated proportional control, the Toddler must be a bit more discrete. It is possible to take proportional steps but the accuracy of the Toddler's movements minimizes the effect of minor changes to movements. On the other hand, the Boe-Bot can move one or both of its wheels a fraction of an inch in subsecond times. A Toddler step can take as long as a second.

As it turns out, the Toddler's IR range finders work well for tracking another Toddler. The range results are in discrete values and the number is not large. If it were, then the

values would have to be converted down to this level of gradation that is manageable. It is then simply a matter of choosing the appropriate step type and magnitude.

The Toddler is a difficult target for another Toddler to locate with its many facets. To improve the detection using the IR sensors, the target Toddler should have a white box placed around it. This can be made of paper or cardboard and it can be affixed to the Toddler's frame using tape or other means. The box should start about where the base of the Toddler's central body contains the servos and can extend to just above the circuit board. The IR sensors can be angled down slightly so they will detect the central portion of the box at a distance of about a foot. The box should not impede the foot movement or the servos and it can extend out from the Toddler by as much as a few inches. It should not be too heavy or large so as to significantly change the center of gravity forcing adjustments in walking behavior.

Although these changes are not absolutely required for one Toddler to follow another, they will improve the overall system performance. Also, the roaming area needs to be free of obstacles and walls otherwise the Toddler that will be following may detect these obstacles instead. There is no checking in the program to determine if the object detected is remaining stationary although it is possible to modify the program to do so.

### **Programming the Toddler Shadow Walker**

Program Listing 7.3 uses additional branch and lookup statements to adjust the Toddler's position based on the range finder results. The movements are designed to aim the Toddler at the object it is following, usually another Toddler, and to keep that object at a discrete distance.

**Your Turn**

- √ Run Program Listing 7.3.
- √ Point the Toddler at an 8 ½ x 11” sheet of paper held in front of it as though it’s a wall-obstacle. The Toddler should move about trying to maintain an average distance from the object.
- √ Try moving the paper so it rotates about the Toddler. The Toddler should rotate with it.
- √ Try using the sheet of paper to lead the Toddler around. The Toddler should follow it.

```

' -----[ Title ]-----
' Toddler Program 7.3: Shadow Walker
' Follows another Toddler with a piece of paper on his backside
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

StrideServo    CON    12          ' Stride servo on P12
TiltServo      CON    13          ' Tilt servo on P13

left_pin       CON    4
right_pin      CON    15

left_in        VAR    IN11
right_in       VAR    IN14

' -----[ Constants ]-----

MoveDelay      CON    25          ' in microseconds

TiltStep       CON    20          ' TiltServo step size
StrideStep     CON    20          ' StrideServo step size

RightTilt      CON    630         ' Tilt limits
CenterTilt     CON    750
LeftTilt       CON    870

RightStride    CON    650         ' Stride limits
CenterStride   CON    750
LeftStride     CON    850

```

```

' -----[ Variables ]-----
FigureLoop      VAR      Nib
MoveLoop        VAR      Byte      ' Loop for repeat movements
MoveLoopLimit   VAR      Byte
SubMoveLoop     VAR      Byte      ' Loop for repeat submovements
SubMoveLoopLmt VAR      Byte
Pulses          VAR      Word      ' Pulse variable
CurrentTilt     VAR      Word
CurrentStride   VAR      Word
NewValue        VAR      Word
Dx              VAR      Pulses
Mx              VAR      Word
MxCurrent       VAR      Word
Sx              VAR      Word
SxCurrent       VAR      Word
counter         VAR      Nib      ' For...next loop index variable.
l_values        VAR      Mx      ' store R sensor vals
r_values        VAR      Sx      ' store L sensor vals
l_IR_freq       VAR      MxCurrent ' stores L IR frequencies.
r_IR_freq       VAR      SxCurrent ' stores R IR frequencies

' -----[ Movement Support Codes ]-----
'
' The following state tables are lists of movement state numbers.
' A xx indicates the end of a list.
' These are used with the Movement routine.

TL              CON      0
TC              CON      1
TR              CON      2

SL              CON      3
SC              CON      4
SR              CON      5

xx              CON      255

' -----[ EEPROM Data ]-----
'
' ----- Movement Value Tables -----
' These can be used with the Movement routine.
' The tables can contain Basic Movement Codes.
'

```

```

' Note: ALL movement tables must be in this section

TurnLeftForward      DATA      1, bLeftTurn, bForward, xx
TurnRightForward     DATA      1, bRightTurn, bForward, xx

PivotLeftForward     DATA      1, bPivotLeft, bForward, xx
PivotRightForward    DATA      1, bPivotRight, bForward, xx

BackwardPivotLeft    DATA      1, bBackward, bPivotLeft, xx
BackwardPivotRight   DATA      1, bBackward, bPivotRight, xx

Forward2             DATA      2, bForward, xx
Backward2            DATA      2, bBackward, xx

' ----- Basic Movement Codes -----
'
' Used in Movement tables.
' Referenced below using LOOKUP statement.

bFinish      CON      0
bForward     CON      1
bBackward    CON      2
bLeftTurn    CON      3
bRightTurn   CON      4
bPivotLeft   CON      5
bPivotRight  CON      6

' ----- Basic Movement Tables -----
'
' These tables can contain Movement Support Codes.

BasicMovements  CON      Forward
Nop             DATA    1, xx

Forward         DATA    1, TR, SL, TL, SR, xx
Backward        DATA    1, TR, SR, TL, SL, xx

LeftTurn        DATA    1, TL, SR, TC, SL, TL, SR, TR, SL, xx
RightTurn       DATA    1, TR, SL, TC, SR, TR, SL, TL, SR, xx

PivotLeft       DATA    3, TL, SR, TC, SL, TR, SR, TC, SL, xx
PivotRight      DATA    3, TR, SL, TC, SR, TL, SL, TC, SR, xx

Finish         DATA    1, TR, SC, TC, xx

'----- Movement LOOKUP entries -----
'
' These constants should reference the appropriate movement table.

```

```

' The constant syntax is lxry where x and y indicate the range from the
' left and right sensor respectively. A zero value indicates nothing
' is within range while a 5 indicates an object is within inches.
' In general, a 3 will be the closest desirable distance.

10r0    CON    Forward
10r1    CON    TurnRightForward
10r2    CON    PivotRightForward
10r3    CON    PivotRight
10r4    CON    RightTurn
10r5    CON    BackwardPivotRight

11r0    CON    PivotLeftForward
11r1    CON    Forward
11r2    CON    PivotRightForward
11r3    CON    PivotRight
11r4    CON    PivotRight
11r5    CON    BackwardPivotRight

12r0    CON    TurnLeftForward
12r1    CON    TurnLeftForward
12r2    CON    Forward
12r3    CON    Nop
12r4    CON    PivotRight
12r5    CON    BackwardPivotRight

13r0    CON    PivotLeft
13r1    CON    PivotLeft
13r2    CON    Nop
13r3    CON    Nop
13r4    CON    Nop
13r5    CON    BackwardPivotRight

14r0    CON    BackwardPivotLeft
14r1    CON    BackwardPivotLeft
14r2    CON    PivotLeft
14r3    CON    Nop
14r4    CON    Backward
14r5    CON    Backward

15r0    CON    BackwardPivotLeft
15r1    CON    BackwardPivotLeft
15r2    CON    BackwardPivotLeft
15r3    CON    BackwardPivotLeft
15r4    CON    Backward
15r5    CON    Backward

'----- Initialization -----

OUTPUT 2                                ' Declare outputs.
OUTPUT left_pin

```



```

OUTPUT right_pin

FREQOUT 2,500,3000          ' Beep at startup.

GOSUB ResetCC

'----- Main Routine -----
main:                      ' Main routine

  GOSUB check_sensors      ' Distance values for each sensor

  'debug "l",dec l_values,"r", dec r_values,cr

  BRANCH l_values, [left0,left1,left2,left3,left4,left5]

left0:
  LOOKUP r_values, [l0r0,l0r1,l0r2,l0r3,l0r4,l0r5],Mx
  GOTO main_movement

left1:
  LOOKUP r_values, [l1r0,l1r1,l1r2,l1r3,l1r4,l1r5],Mx
  GOTO main_movement

left2:
  LOOKUP r_values, [l2r0,l2r1,l2r2,l2r3,l2r4,l2r5],Mx
  GOTO main_movement

left3:
  LOOKUP r_values, [l3r0,l3r1,l3r2,l3r3,l3r4,l3r5],Mx
  GOTO main_movement

left4:
  LOOKUP r_values, [l4r0,l4r1,l4r2,l4r3,l4r4,l4r5],Mx
  GOTO main_movement

left5:
  LOOKUP r_values, [l5r0,l5r1,l5r2,l5r3,l5r4,l5r5],Mx

main_movement:
  GOSUB Movement
  GOTO main                ' Infinite loop.

'----- Subroutine(s) -----

check_sensors:

  l_values = 0              ' Set distances to 0.
  r_values = 0
  ' Take 5 measurements for distance at each IR pair.  If you fine tuned

```

```

' frequencies in Activity #2, insert them in the lookup tables.

FOR counter = 0 TO 4
  check_left_sensors:
    LOOKUP counter, [37500,38250,39500,40500,41000],l_IR_freq
    FREQOUT left_pin,1,l_IR_freq
    l_values.LOWBIT(counter) = ~left_in

  check_right_sensors:
    LOOKUP counter, [37500,38250,39500,40500,41000],r_IR_freq
    FREQOUT right_pin,1,r_IR_freq
    r_values.LOWBIT(counter) = ~right_in

NEXT

l_values = NCD l_values          ' Value for distance depends on MSB
r_values = NCD r_values

RETURN

' ----- Movement: Move feet using DATA table referenced by Mx -----
'
' Input: Mx = movement table index, table ends in xx
'       or
'       Mx = submovement table index, table ends in xx
'
' Note: All submovement tables come after the movement tables in this file.

Movement:
  IF Mx < BasicMovements THEN SetupMovement

  MxCurrent = Mx                ' setup to use submovement table
  MoveLoopLimit = 1
  GOTO StartMovement

SetupMovement:
  READ Mx, MoveLoopLimit        ' read movement table repeat count
  MxCurrent = Mx + 1

StartMovement:
  FOR MoveLoop = 1 TO MoveLoopLimit
    Mx = MxCurrent              ' Mx = start of movement table

' debug hex Mx, " Movement ", dec MoveLoop, " of ", dec MoveLoopLimit, cr

    IF Mx < BasicMovements THEN MovementLoop
      ' skip if movement table
      SxCurrent = Mx            ' SxCurrent = submovement index
      GOTO StartSubMovement    ' enter middle of loop

  MovementLoop:

```

```

    READ Mx, SxCurrent          ' read next submovement byte
    Mx = Mx + 1
    IF SxCurrent = xx THEN MovementDone
                                ' skip if end of list
    'debug " ", hex SxCurrent, " movement",cr
    LOOKUP SxCurrent, [Finish,Forward,Backward,LeftTurn,RightTurn,
    PivotLeft,PivotRight],SxCurrent
                                ' lookup submovement table index
StartSubMovement:
    READ SxCurrent, SubMoveLoopLmt
                                ' start executing submovement table
                                ' read submovement table repeat count
    SxCurrent = SxCurrent + 1

    FOR SubMoveLoop = 1 TO SubMoveLoopLmt
        Sx = SxCurrent
    'debug " ", hex Sx, " submovement ", dec SubMoveLoop, " of "
    'debug dec SubMoveLoopLmt,cr

        SubMovementLoop:
            READ Sx, Dx          ' read next submovement action
            Sx = Sx + 1
            IF Dx = xx THEN SubMovementDone
                                ' skip if end of list

            GOSUB DoMovement    ' execute movement
            GOTO SubMovementLoop

        SubMovementDone:
    NEXT
    IF Mx < BasicMovements THEN MovementLoop
                                ' exit if submovement table
    MovementDone:
    NEXT
    RETURN

    DoMovement:
    'debug " ", dec Dx, " action",cr
    BRANCH Dx, [TiltLeft,TiltCenter,TiltRight,StrideLeft,
    StrideCenter,StrideRight]
                                ' will fall through if invalid index
    RETURN

' ---- Movement routines can be called directly ----

TiltLeft:
    NewValue = LeftTilt
    GOTO MovementTilt

TiltCenter:
    NewValue = CenterTilt
    GOTO MovementTilt

```

```

TiltRight:
  NewValue = RightTilt

MovementTilt:
  FOR Pulses = CurrentTilt TO NewValue STEP TiltStep
    PULSOUT TiltServo, Pulses
    PULSOUT StrideServo, CurrentStride
    PAUSE MoveDelay
  NEXT

  CurrentTilt = NewValue
  RETURN

StrideLeft:
  NewValue = LeftStride
  GOTO MovementStride

StrideCenter:
  NewValue = CenterStride
  GOTO MovementStride

StrideRight:
  NewValue = RightStride

MovementStride:
  FOR Pulses = CurrentStride TO NewValue STEP StrideStep
    PULSOUT TiltServo, CurrentTilt
    PULSOUT StrideServo, Pulses
    PAUSE MoveDelay
  NEXT

  CurrentStride = NewValue
  RETURN

' ----- Move feet to initial center position -----

ResetCC:
  CurrentTilt = CenterTilt
  CurrentStride = CenterStride

  FOR Pulses = 1 TO 100 STEP StrideStep
    PULSOUT TiltServo, CenterTilt
    PULSOUT StrideServo, CenterStride
    PAUSE MoveDelay
  NEXT

DoReturn:
  RETURN

```

### How the Shadow Walker Program Works

The first thing the main routine does is call the `check_sensors` subroutine. After the `check_sensors` subroutine is finished, `l_values` and `r_values` each contain a number corresponding to the zone in which an object was detected for both the left and right IR pairs.

```
main:
    GOSUB check_sensors
```

The next line of code jumps to one of many `LOOKUP` statements. The `BRANCH` statement uses the status of the left IR sensor while the `LOOKUP` statements use the status of the right IR sensor. These set the `Mx` variable with the table index for the movement to be performed by the `Movement` routine.

```
Branch l_values, [left0, left1, left2, left3, left4, left5]

left0:
    LOOKUP r_values, [l0r0, l0r1, l0r2, l0r3, l0r4, l0r5], Mx
    GOTO main_movement

left1:
    LOOKUP r_values, [l1r0, l1r1, l1r2, l1r3, l1r4, l1r5], Mx
    GOTO main_movement

left2:
    LOOKUP r_values, [l2r0, l2r1, l2r2, l2r3, l2r4, l2r5], Mx
    GOTO main_movement

left3:
    LOOKUP r_values, [l3r0, l3r1, l3r2, l3r3, l3r4, l3r5], Mx
    GOTO main_movement

left4:
    LOOKUP r_values, [l4r0, l4r1, l4r2, l4r3, l4r4, l4r5], Mx
    GOTO main_movement

left5:
    LOOKUP r_values, [l5r0, l5r1, l5r2, l5r3, l5r4, l5r5], Mx

main_movement:
    GOSUB Movement
```

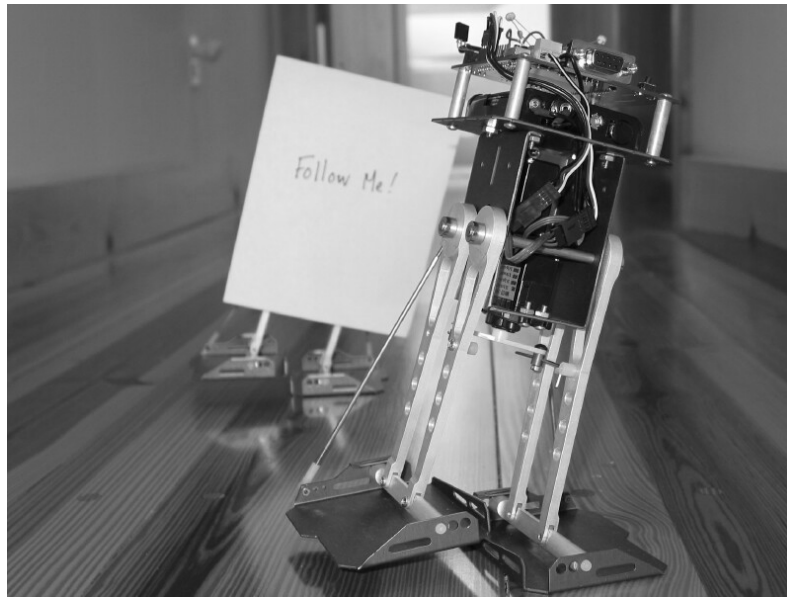
7

The values used in the `LOOKUP` statement are defined near the start of the program using `CON` constant definitions. While it is possible to put these values in the `LOOKUP` statement,

this makes the statements long. It also makes it difficult to see what action is performed in a particular state. The constant definitions provide a way to do this. It is now easy to correlate a particular state such as `13r3` with a particular movement, in this case a `nop` or no movement. Likewise, `15r5` indicates that the Toddler is immediately in front of an obstacle and `10r0` indicates the Toddler has not located an obstacle within its range.

Program control is returned to the `main:` label after the movement has been performed, and the loop repeats itself.

## CHALLENGES



7

**Figure 7-5:** One Toddler Follows Another Toddler

Figure 7.5 shows a lead Toddler followed by a shadow Toddler. The lead Toddler could run any of the prior programs provided the speed is slower (increase the **PAUSE** values or decrease the **STEP** values) and the shadow Toddler is running Program Listing 7.3: Shadow Walker. Proportional control makes the shadow Toddler a very faithful follower. One lead Toddler can string along a chain of 2 or 3 Toddlers. Just add a 4" x 4" paper to the lead Toddler's backside.

- √ If you are part of a class, mount paper panel on the back of the lead Toddler as shown in Figure 7.5.
- √ If you are not part of a class (and only have one Toddler) the shadow Toddler will follow a piece of paper or your hand just as well as it follows a lead Toddler.

- √ The Shadow Toddler should be running Program Listing 7.3 without any modifications.
- √ With both Toddlers running their respective programs, place the shadow Toddler behind the lead Toddler. The shadow Toddler follows at a fixed distance, so long as it is not distracted by another object such as a hand or a nearby wall.



## Chapter #8: Real-Time Programming with Bumpers and Infrared

---

*Note to reader: This chapter requires the Toddler Toes Kit (#27312) from Parallax.*

### WHAT IS MEANT BY REAL TIME?

Real time programming is one of those terms that can mean different things at the same time. In plain language, it is code that allows the computer system to keep up with what is happening in the world around it – WHILE it is happening. Like in everything else, there are degrees of “real time”. Programming a system to respond to events happening quickly is more difficult than if they happen more slowly. Programming events to happen at a very precise time can also be challenging – even if they don’t happen at high speed.

Let’s look at a simple example.

You want to flash 5 LEDs. You can determine if they need to be on or off by testing switch inputs. It’s an easy task.

1. Turn off all the LEDs to start
2. Test each switch input and turn on the LED if the switch requires it.
3. Pause for ½ second – or however long you want the LEDs on.
4. Turn off all the LEDs
5. Pause for ½ second – or however long you want the LEDs off.
6. Go to #2 and start over.

You probably ignored how long it takes to test the inputs in step #2 before running the **PAUSE** in step #3. What if the program were required to flash the LEDs at precisely a 1 second rate (+- 0.001%)? Ignoring the calculation times can lead to disaster in that case. Not only would you have to measure or calculate the instruction times to correct the duration of the **PAUSE** but also you’d have to make sure that all possible cases of switch inputs take the same amount of time to compute.

Now, what if the switches only make contact for 20 milliseconds? Your code would have to constantly be scanning to see if the switch was making contact and remember it. The scanning has to happen while still keeping track of the flashing LEDs. If you had a loop to check the switches, you’d have to complete that loop more than once every 20

milliseconds. The BASIC Stamp makes it so easy to turn on or off LEDs that it probably still isn't a big problem. What if you had to do some calculations before deciding to turn the LED on or off? And what if those calculations take 30 ms for each LED?

### **A Processor's Got to Know its Own Limitations!**

Actually the processor doesn't care. But it sure helps a programmer to know them. Of course, one way to help in high-speed real time programming is to use a faster processor. But every processor has some limit. Often it isn't price effective to just throw a faster processor at a job. Besides, where's the fun in that? One key to getting the most of any given processor is to use some simple techniques (and sometimes not so simple techniques) to maximize the use of the performance that is there.

In this situation with the BASIC Stamp, more speed adds some help but it would really be nice to do more than one thing at a time.

### **Where's Real Time in Toddler?**


Toddler Program 6.2: Object Detection And Avoidance sensed the outside surroundings using Infrared headlights and decided to where to move accordingly. It responded rather slowly because the sensing was only done between discrete moves. Wouldn't it be better to sense all the time during moves and respond immediately? As mentioned before, Toddler has a more difficult time dealing with sensor inputs because it is walking instead of rolling. In a rolling robot like Boe-Bot, you can back up or turn at any time. Toddler can't. If both feet are on the ground, it may first have to lift a leg before trying to move. This adds complexity to the program compared to a roller, but it can still be done. Better yet, it adds some personality to your Toddler too.

In this experiment we will use the Toddler Toes and the infrared object detection circuit from prior experiments to demonstrate real-time programming on the Toddler.

## ACTIVITY #1: BUILDING AND TESTING TODDLER TOES AND INFRARED CIRCUIT

### Parts Required

- (1) Piezoelectric speaker
- (2) Shrink wrapped IR LEDs
- (2) IR detectors
- (1) Set of Toddler Toes (Parallax #27312, not included in this kit)
- Optional: (2) LEDs and (2) 220 ohm resistors for feedback purposes
- (misc) wires

	<b>Toddler Toes</b> kit (#27312) is not included in the stock Toddler Kit (#27310 or #27311) because it would have raised the retail price, prohibiting more robot enthusiasts from purchasing the kit. You can make your own toes using common materials.		
	This is what is included in the Parallax Toddler Toes Kit (#27312):		
	Parallax Part Number	Quantity	Description
	550-27312	1	Twinkle Toes PCB, right side
	550-27313	1	Twinkle Toes PCB, left side
	700-00002	6	4/40 3/8" machine screw
	700-00003	8	4/40 nut
	720-27312	6	Bumper wire (extras for possible mistakes)
	805-00002	2	14" servo extension cable

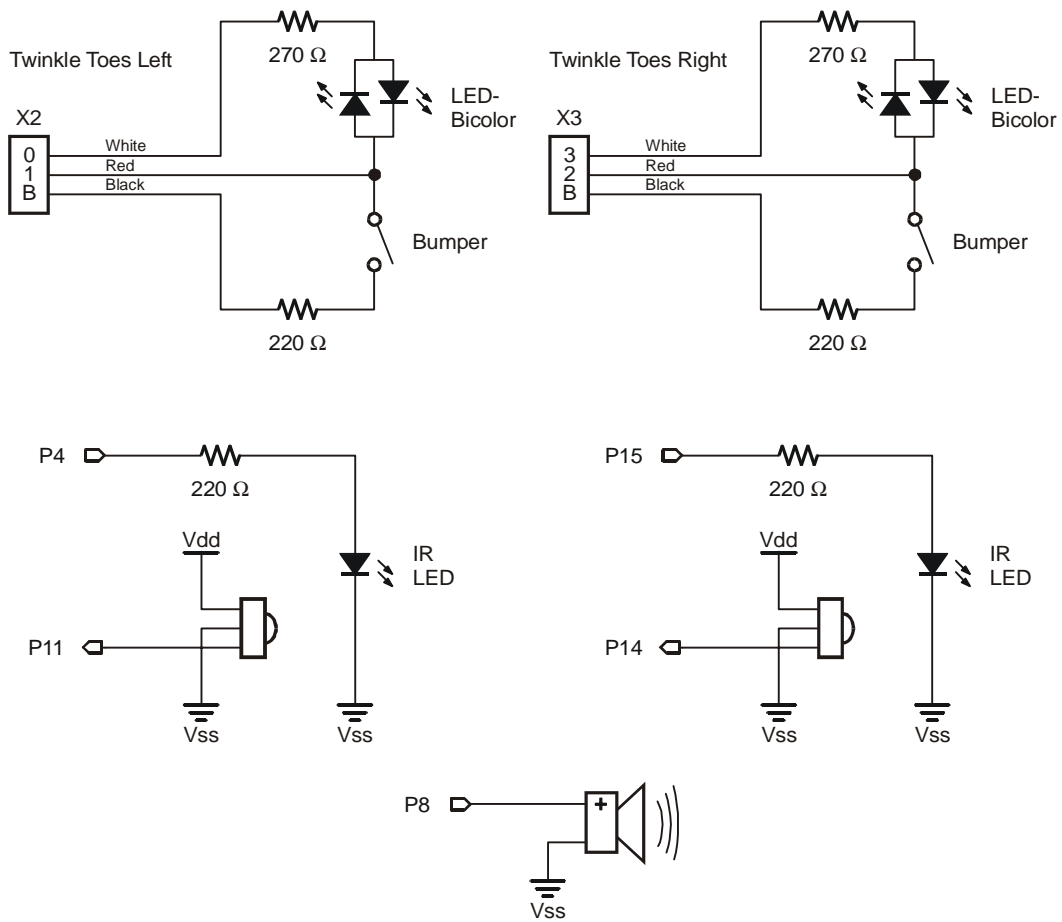
8

### **Assembling the Toddler Twinkle Toes**

Figure 8.1 shows the schematic for the experiment.

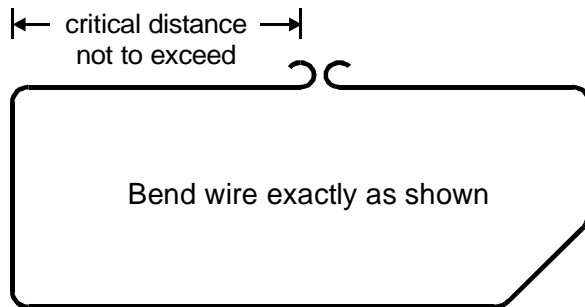
Bending the Wires: Bend the wire to match the outline of the bumper in Figure 8-2. You will need to use needle nose pliers to do this job.

The two curled ends need to make a 50% to 75% closed circle to keep from sliding off the screw. Try to keep the wire flat while making these bends. The distance from one of the curls to the outside bend is not to be exceeded. Follow the template exactly and the bumpers will not hit each other.



**Figure 8-1:** Real-Time Programming with Bumpers and Infrared. You may optionally add two green or red LEDs to pins 5 and 6 on the breadboard to provide visual feedback for the infrared detectors.

You have six wires with which to experiment. You only need two good bumpers so don't worry if the first ones don't come out quite right. And if you really destroy all of your wires you can call Parallax for a few more.



**Figure 8-2:** Wire bending template

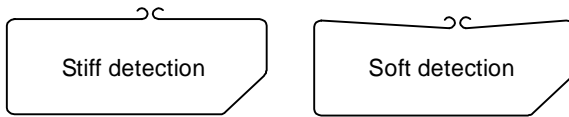
Fine Tuning the Wire Tension: Put the 3/8" screw through the center hole of each Twinkle Toes and secure tightly with a nut on top.

Attach the bumper hook ends over the screw you just mounted in the center of each board. The beveled corner goes on the side with the LED. Slip the wire in-between the prongs (two-post header pins) at each end of the board.

8

The bumper should be touching the pin closest to the front of the board. When pressing the bumper from the front the wire will touch the pin farthest from the front of the board. The pressure should feel very light to trip the bumper and will need to be adjusted until it is lightly springy.

Figure 8-3 shows how to adjust the bumper wire to be less stiff. For a final adjustment, take the bumper off, adjust and retest the tension. Once both sides of the bumper trip with a soft bump, secure with the locking nut. Do not tighten on the wire; the wire needs to be able to move freely around the screw.

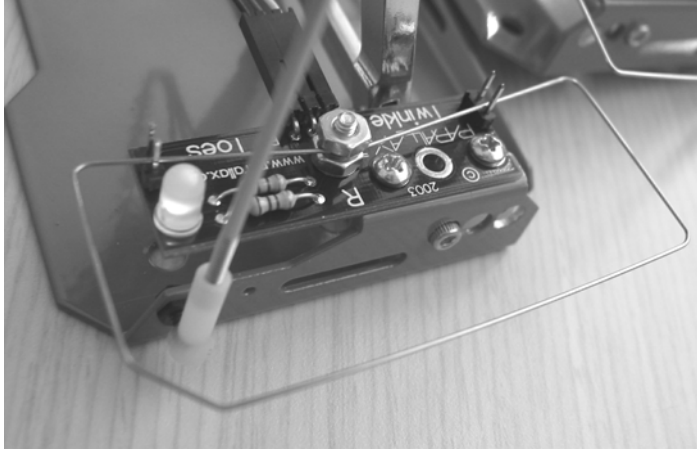


**Figure 8-3:** Bumper Tensioning

Figure 8-4 and Figure 8-5 show the final installation of the Twinkle Toes.



**Figure 8-4:** Installed Twinkle Toes



**Figure 8-5:** Twinkle Toes include the pushbutton resistors and bi-color LED.

Mounting Twinkle Toes Disconnect the tilt rods from the Toddler's left foot (the same side as the power switch). Mount the Twinkle Toes with two 1/4" screws (the third hole is only used if the ankle is mounted to the inside which would block one of the holes).

Make sure the bumper does not extend past the inside edge of the foot. If it does bend the wire or make a new bumper. One way to find out if the bumper is going to touch the opposite foot is to run one of the early basic walking programs in this text.

Reattach the control rod to the foot then mount the right sensor the same way and make sure it also does not extend past the inside edge of the foot. Repeat for the right foot.

Connect the cables Connect the cables from the sensors to X2 and X3 of the Toddler Board making sure the black to black connection (B to B) is maintained.

### Testing the Toddler Toes

Let's test the Toddler Toes circuit before we proceed to do anything else.

```

' -----[ Title ]-----
' Toddler Program 8.1 Twinkle Toes Test.bs2
' This program tests twinkle toes bumpers
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Constants ]-----

ctrlLEDS      CON      %1111      ' Define DIRS as constants
prepBumper    CON      %1001

bothGreen     CON      %0110      ' OUTa register
bothRed       CON      %1001
rightRed      CON      %0101
leftRed       CON      %1010
bothLEDSOff   CON      %1001

' -----[ Main Code ]-----

Forward:
  DIRS = ctrlLEDS
  OUTS = bothGreen
  PAUSE 20

Check_Bumpers:
  DIRS = prepBumper
  OUTS = bothLEDSOff
  DEBUG Home, "Right = ", BIN1 IN1, " Left = ", BIN1 IN2
  IF (IN1 = 0) AND (IN2 = 0) THEN BothOn
  IF IN1 = 0 THEN Right
  IF IN2 = 0 THEN Left
  GOTO Forward

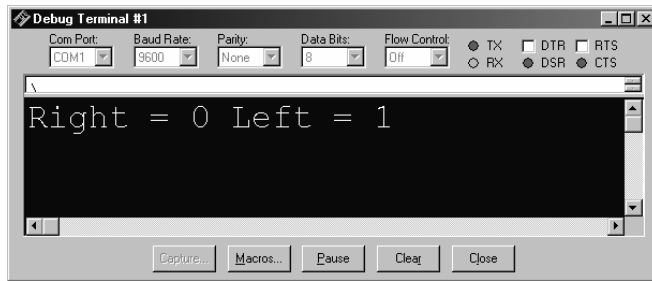
BothOn:
  DIRS = ctrlLEDS
  OUTS = bothRed
  PAUSE 200
  GOTO Forward

Right:
  DIRS = ctrlLEDS
  OUTS = rightRed
  PAUSE 200
  GOTO Forward

Left:
  DIRS = ctrlLEDS
  OUTS = leftRed
  PAUSE 200
  GOTO Forward

```





**Figure 8-6:** Toddler Toes Debug Test. Pushing the bumper should result in a “0” on the I/O pin.

If the Toes are properly installed and wired then you will be able to activate them and view the status of each bumper in the Debug Terminal as shown in Figure 8-6. If it works then proceed. If it does not work then double-check your three-conductor wire connectors and the wiring. The Twinkle Toes should flash green unless you press a bumper, then you will see a green-red intermittent flash.

### Testing the Toddler Toes and Infrared Object Detection Circuit Together

Now make sure the infrared emitters and detectors are plugged in the front of the Toddler board. Then, run the following program to test the two circuits together.

```
' -----[ Title ]-----
' Toddler Program 8.2 Infrared and Twinkle Toes Test.bs2
' This program tests infrared sensors and twinkle toes bumpers together
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      PIN    13      ' tilt servo on X7
StrideServo    PIN    12      ' stride servo on X6

LeftIRLED      PIN     4      ' left IRLED
RightIRLED     PIN    15      ' right IRLED
LeftIR_Sensor  PIN    11      ' left IR sensor input
RightIR_Sensor PIN    14      ' right IR sensor input

LeftBumper     PIN     2
RightBumper    PIN     1

Spkr           PIN     8      ' beeper pin

' -----[ Constants ]-----
```

```

MoveDelay      CON      17          ' to make up 20 ms servo loop
CenterTilt     CON      750
CenterStride   CON      750

prepBumper     CON      %1001
bothLEDsOff    CON      %1001

' -----[ Variables ]-----

Counter        VAR      Nib
Temp           VAR      Bit

' -----[ Initialize ]-----

ResetFeet:

  FOR Counter = 1 TO 15          ' center feet for .3 sec
    GOSUB Move_It              ' currentMove is only a nibble.
    PAUSE MoveDelay            ' make up 20 ms servo refresh time
  NEXT

Test_Speaker:
  FREQOUT Spkr, 2000, 3000      ' program start/restart signal.

' -----[ Main Code ]-----

Main_Program:

DO

  FREQOUT LeftIRLED,1,38500      ' pulse left IRLED.
  IF (LeftIR_Sensor = 0) THEN    ' check for IR hit on left
    DEBUG " IR left "
  ELSE
    DEBUG " "
  ENDIF

  FREQOUT RightIRLED,1,38500     ' repeat for the right IR pair.

  IF (RightIR_Sensor = 0) THEN   ' check for IR hit on right
    DEBUG " IR Right "
  ELSE
    DEBUG " "
  ENDIF

  FREQOUT RightIRLED,1,38500     ' repeat for the right IR pair.

Do_Bumpers:
  DIRS = prepBumper
  OUTS = bothLEDsOff

```

```

IF (LeftBumper = 0) THEN          ' check for bumper hit on left
  DEBUG " bump left "
ELSE
  DEBUG "          "
ENDIF

IF (RightBumper = 0) THEN        ' check for bumper hit on right
  DEBUG " bump Right ", CR
ELSE
  DEBUG "          ", CR
ENDIF

LOOP

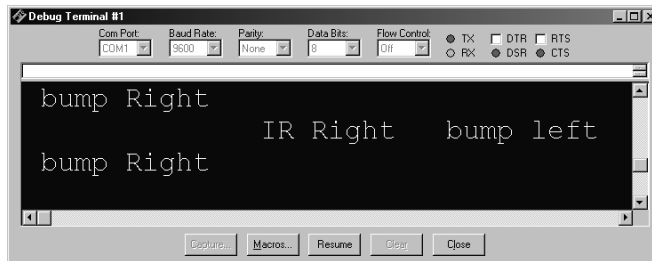
END

' -----[ Subroutines ]-----

Move_It:
  PULSOUT TiltServo,  CenterTilt
  PULSOUT StrideServo, CenterStride
  RETURN

```

This program will allow you to test the I/O connections you just made.



**Figure 8-7:** Toddler Toes and Infrared Debug Test. The sensors that detect an object will be shown in the Debug Terminal.

**ACTIVITY #2: FINITE STATE MACHINE REAL-TIME WALKER**

Now you are ready to enter and run Program Listing 8.3, “Toddler Program 8.3 Finite State Machine Walker.bs2”. This program also uses an optional four indicator LEDs on P5 – P8. If you connect the LEDs be sure to use four resistors between the BASIC Stamp I/O pin and LED.

```
' -----[ Title ]-----
' Toddler Program 8.3: Finite State Machine Walker
' Real time servo programming and sensor integration lets Toddler walk
' until infrared or Toddler Toes detect an object. To walk with bumpers
' only simply remove both infrared emitters on P4 and P15.
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      PIN    13      ' tilt servo on X7
StrideServo    PIN    12      ' stride servo on X6

LeftIRLED      PIN    4        ' left IRLED
RightIRLED     PIN    15       ' right IRLED
LeftIRSensor   PIN    11       ' left IR sensor input
RightIRSensor  PIN    14       ' right IR sensor input

RightBumper    PIN    1
LeftBumper     PIN    2

LeftVisionLED  PIN    5        ' left vision display LED
RightVisionLED PIN    6        ' right vision display LED
Spkr           PIN    8        ' beeper pin

' -----[ Constants ]-----

MoveDelay      CON    1        ' code already using 19 ms of 20
TiltStep       CON    5        ' TiltServo step size
StrideStep     CON    5        ' StrideServo step size

RightTilt      CON    610      ' tilt limits
CenterTilt     CON    750
LeftTilt       CON    880

RightStride    CON    625      ' stride limits
CenterStride   CON    750
LeftStride     CON    875

ctrlLEDs       CON    %1111    ' define DIRs as constants to
```

```

prepBumper      CON      %1001      ' simplify LED/bumper setups

bothGreen       CON      %0110      ' OUTa register sets red/green
bothRed         CON      %1001      ' status on P0-P3 for Twinkle Toes
rightRed        CON      %0101      ' object detection states
leftRed         CON      %1010
bothLEDsOff     CON      %1001

Fwd             CON      0          ' codes to pick movement table
PivL           CON      1          ' set newMove to one of these
PivR           CON      2          ' values and New_Movemt will LOOKUP
BumpL          CON      3          ' the right movement table
BumpR          CON      4

' -----[ Variables ]-----

doneTiltFlag    VAR      Bit        ' flag: 1 = tilt reached new value
doneMoveFlag    VAR      Bit        ' flag: 1 = servos hit new values
rightIR_Flag    VAR      Bit        ' flag: 1 = something on right side
leftIR_Flag     VAR      Bit        ' flag: 1 = something left side
rightBumpFlag   VAR      Bit        ' flag: 1 = bumper hit right side
leftBumpFlag    VAR      Bit        ' flag: 1 = bumper hit left side

sensors         VAR      Nib        ' lower 2 bits of the sensors var
                ' used to store IR detector values

currentMove     VAR      Nib
newMove         VAR      Nib

Mx              VAR      Word       ' index for movement tables
MxOffset        VAR      Byte       ' added to Mx for index
bMovmnt         VAR      Byte       ' table value for lookup movement

currentTilt     VAR      Word
currentStride   VAR      Word
newTilt         VAR      Word
newStride       VAR      Word

' -----[ EEPROM Data ]-----
'
' These are actual values saved in the Basic Movement tables.

TL              CON      0          ' use lower nibble for these
TC              CON      1
TR              CON      2
SL              CON      3
SC              CON      4
SR              CON      5
xx              CON      255       ' table end code

' ----- Basic Movement Tables -----
'

```

```

' These tables contain Basic Movements consisting of bytes containing
' above Basic Movement Codes to describe movement sequences.
' An xx indicates the end of a list.
' PivotLeft and PivotRight aren't entered at the start of their tables

Forward      DATA    TR, SL, TL, SR, xx
PivotLeft    DATA    TR, SL, TL, SR
              DATA    TC, SL, TL, SR, xx
PivotRight   DATA    TR, SL, TL, SR
              DATA    TR, SL, TC, SR, xx
RBumper      DATA    SL, TR, SR, TC, SL, TL, SR, xx
LBumper      DATA    SR, TL, SL, TC, SR, TR, SL, TL, SR, xx

' -----[ Initialize ]-----

GOSUB Clr_Vision           ' reset vision LEDs and flags
DIRS = ctrlLEDS           ' setup green LEDs for Forward
OUTS = bothGreen

ResetFeet:
  newTilt = CenterTilt
  newStride = CenterStride
  currentTilt = CenterTilt
  currentStride = CenterStride

  FOR currentMove = 1 TO 15           ' center feet for .3 sec
    GOSUB Move_It                     ' currentMove is only a nibble.
    PAUSE 17                          ' make up 20 ms servo refresh time
  NEXT

  'DEBUG "Forward = ", DEC Forward, CR
  'DEBUG "PivotLeft = ", DEC PivotLeft, CR,
  'DEBUG "PivotRight = ", DEC PivotRight, CR
  'DEBUG "RBumper = ", DEC RBumper, cr, "LBumper = ", DEC LBumper, CR

  doneTiltFlag = 1
  doneMoveFlag = 1
  sensors = 0
  MxOffset = 0

  FREQOUT Spkr, 2000, 3000           ' program start/restart signal
  rightBumpFlag = 0
  leftBumpFlag = 0

  currentMove = 15                   ' invalid value to assure start
  ' newMove = Fwd                       ' for testing single moves -
  ' newMove = PivL                       ' comment out GOSUBs to vision
  ' newMove = PivR                       ' and bump or the value might be
  ' newMove = BumpL                       ' overwritten
  ' newMove = BumpR

```

```

' -----[ Main Code ]-----
Main_Program:
DO
  GOSUB Do_Vision           ' look for obstacles with IR
  GOSUB Do_Bumper          ' check for obstacles with bumpers
  GOSUB New_Movement       ' generates next move from table
  GOSUB Do_Movement        ' gives servo pulses out
  PAUSE MoveDelay          ' makes up 20 ms servo timing loop
LOOP
END

' -----[ Subroutines ]-----

Do_Vision:

  FREQOUT LeftIRLED,1,38500      ' pulse left IRLED.
  sensors.BIT0 = LeftIRSensor    ' store IR detector output in RAM

  FREQOUT RightIRLED,1,38500     ' repeat for the right IR pair.
  sensors.BIT1 = RightIRSensor

  ' Check if currently doing an infrared move

  IF ((leftIR_Flag = 1) & (rightIR_Flag = 1)) THEN See_Both
  IF (leftIR_Flag = 1) THEN See_Left   ' is left only?
  IF (rightIR_Flag = 1) THEN See_Right ' is right only?

  ' Load IR detector output values into the lower 2 bits of the sensors
  ' variable, storing a number between 0 and 3 that the BRANCH command
  ' can execute the appropriate routine.

  BRANCH sensors, [See_Both,See_Right,See_Left,See_None]

See_Both:
  newMove = PivR
  rightIR_Flag = 1           ' flag: 1 = something on right side
  leftIR_Flag = 1           ' flag: 1 = something left side
  HIGH LeftVisionLED        ' left vision sequence display LED
  HIGH RightVisionLED       ' right vision sequence display LED
  RETURN

See_Right:
  newMove = PivL
  rightIR_Flag = 1           ' flag: 1 = something on right side
  HIGH RightVisionLED       ' right vision sequence display LED
  RETURN

See_Left:
  newMove = PivR
  leftIR_Flag = 1           ' flag: 1 = see something left side

```

```

HIGH LeftVisionLED          ' left vision sequence display LED
RETURN

See_None:
  newMove = Fwd
  RETURN

Do_Bumper:
  DIRS = prepBumper
  OUTS = bothLEDsOff

  ' Bumpers on each foot. A hit causes Toddler to back and turn.
  ' Code assumes leg that hit is on the leg that is moving
  ' (while leaning on the other leg).
  ' A hit backs up the hit leg right away then steps back 1 step
  ' and does a pivot away.

  IF (leftBumpFlag = 1) THEN Bump_LeftA
  IF (rightBumpFlag = 1) THEN Bump_RightA

  ' check for old bumper hit on left
  ' check for old bumper hit on right

  IF (LeftBumper = 0) THEN Bump_Left    ' check for bumper hit on left
  IF (RightBumper = 0) THEN Bump_Right  ' check for bumper hit on right

  DIRS = ctrlLEDs
  OUTS = bothGreen

  RETURN

Bump_Left:
  'DEBUG "bump left", cr          ' sends only once on change

Bump_LeftA:
  leftBumpFlag = 1              ' flag indicates bump response move
  newMove = BumpL
  DIRS = ctrlLEDs
  OUTS = leftRed
  GOTO Clr_Vision              ' goes to Clr_Vision

Bump_Right:
  'DEBUG "bump right", cr

Bump_RightA:
  rightBumpFlag = 1
  newMove = BumpR
  DIRS = ctrlLEDs
  OUTS = rightRed
  ' falls through to Clr_Vision
  '-----

```



```

Clr_Vision:
    rightIR_Flag = 0          ' reset vision LEDs and flags
    leftIR_Flag = 0          ' flag: 1 = something on right side
    LOW LeftVisionLED        ' flag: 1 = something left side
    LOW RightVisionLED       ' left vision sequence display LED
    RETURN                   ' right vision sequence display LED

'-----

New_Movemt:
    ' sequences for forward motion

    IF (newMove <> currentMove) THEN      ' new DATA if movement changed

        'DEBUG cr, "start NewMovemt; newMove = ", DEC newMove
        'DEBUG ", currentMove = ", DEC currentMove, CR

        doneMoveFlag = 1                  ' stop current move wherever it is
        currentMove = newMove

        ' When customizing the program, here is where to integrate between
        ' moves. You can make the transitions reasonable by using the existing
        ' Mx value and MxOffset to compute the proper MxOffset to start into
        ' the next move.

        IF ((newMove = Fwd) OR (newMove = BumpL) OR (newMove = BumpR)) THEN
            MxOffset = 0                    ' reset MxOffset value for new move
        ENDIF
        ' note: kept old MxOffset value for lead into pivot left or right

        LOOKUP newMove, [Forward,PivotLeft,PivotRight,LBumper,RBumper],Mx
        ' lookup movement table index
    ENDIF

    IF (doneMoveFlag = 1) THEN             ' done moving - lookup new move
        READ (Mx + MxOffset), bMovmnt      ' read next basic move byte
        'DEBUG "Mx = ", HEX Mx, ", MxOffset = ", HEX MxOffset
        MxOffset = MxOffset + 1

        IF (bMovmnt < xx) THEN             ' end code so do new move
            'DEBUG " bMovmnt = ", HEX bMovmnt, " "

            SELECT bMovmnt                  ' set and display movement type
                CASE TL : newTilt = LeftTilt      : 'DEBUG "TL, ", cr
                CASE TC : newTilt = CenterTilt    : 'DEBUG "TC, ", cr
                CASE TR : newTilt = RightTilt     : 'DEBUG "TR, ", cr
                CASE SL : newStride = LeftStride  : 'DEBUG "SL, ", cr
                CASE SC : newStride = CenterStride : 'DEBUG "SC, ", cr
                CASE SR : newStride = RightStride : 'DEBUG "SR, ", cr
            ENDSELECT

            ' fall through if invalid index

```

```

ELSE
  'DEBUG " xx = finished table", CR, CR
  MxOffset = 0           ' finished move sequence so restart
  leftBumpFlag = 0      ' clear out left bumper hit flag
  rightBumpFlag = 0     ' right too
  DIRS = ctrlLEDS
  OUTFS = bothGreen
  GOSUB Clr_Vision      ' reset vision LEDs and flags
  GOSUB Do_Vision       ' since done previous movement,
  GOSUB Do_Bumper       ' figure out new one
  GOTO New_Movemt      ' get new table and offset
ENDIF

ENDIF

'Done_New_Move:
  RETURN                ' ignore if invalid

'-----

Do_Movement:

  doneMoveFlag = 0      ' reset flags
  doneTiltFlag = 0

  IF (currentTilt < newTilt) THEN ' start tilt calculations
    currentTilt = currentTilt + TiltStep ' inc if coming from right tilt
    IF (currentTilt >= newTilt) THEN ' done?
      currentTilt = newTilt ' stop at limit
      doneTiltFlag = 1
    ENDIF
  ELSE
    currentTilt = currentTilt - TiltStep ' coming from left tilt = decrement
    IF (currentTilt <= newTilt) THEN ' done?
      currentTilt = newTilt ' stop at limit
      doneTiltFlag = 1
    ENDIF
  ENDIF

Move_Stride:

  IF (currentStride < newStride) THEN
    currentStride = currentStride + StrideStep
    ' coming from right stride = increment
    IF (currentStride >= newStride) THEN
      currentStride = newStride
      IF (doneTiltFlag = 1) THEN doneMoveFlag = 1
      ' uses doneMoveFlag to increment states.
    ENDIF
  ELSE

```

```

    currentStride = currentStride - StrideStep
    ' coming from left stride = decrement
    IF (currentStride <= newStride) THEN
    ' jump if still moving
    currentStride = newStride
    IF (doneTiltFlag = 1) THEN doneMoveFlag = 1
    ' uses doneMoveFlag to increment states.
    ENDF
  ENDF
Move_It:
  PULSOUT TiltServo,    currentTilt
  PULSOUT StrideServo, currentStride
  RETURN

```

## How the Program Works

### Movement Management

The Basic Movement Tables are similar to what has been used before. First notice the Pivot tables. The pattern for Forward is at the beginning of both pivot entries. This is a way to smoothly go from walking forward to pivoting when Toddler sees something with its IR sensors. It keeps where in the table it was when it saw something walking forward and starts at the same point in the pivot table to get out of the way. Rather than do the same thing for backing and turning on the bumper tables, an assumption is made that the bumper can only be hit while the leg it is attached to is moving forward. It isn't the most general case but it is usually a reasonable assumption.

In the initialization section, **currentMove** is a nibble variable that isn't needed yet for its intended purpose. It is used here as a counter to save variable space.

The **Main** code is straightforward so let's start by looking at the end of subroutines. **Do\_Movement** starts with **currentTilt** and **currentStride** values and ramps the servos to the **newTilt** and **newStride** values. The difference in this subroutine from the examples before is that only one pulse is sent to each servo before **RETURN**. That means the rest of the code will have to run and then come back to **Do\_Movement** every 20 ms to keep the servos running at their optimum timing. Servos will work at longer refresh times but the performance starts to suffer so let's try for 20ms as the goal (in a rolling robot with pre-modified servos a refresh time every 50 ms is acceptable with out a decay in performance). In the previous programs, only a small tight loop had to keep up with the real time requirement. In this program, much more of the program is now real time code.

Since we started at the end, let's continue backward with the **New\_Movement** subroutine. The code starts by comparing **newMove** to **currentMove**. If there is no change it skips the code required to change over to a new move. It then tests to see if the current tilt or stride is finished. Again, if no action is required, skip over it. First rule of real time programming - If you don't absolutely have to do it in the high-speed loop, don't!

In **New\_Movement**, the most usual case is test, test, **RETURN**. It's pretty quick. In the case where the **doneMoveFlag** is 1, it will look up the next item in the active Movement Table. If the current table is finished (**value = xx**), the code gets the next table and starts to run that.

If something caused the **newMove** to be different from **currentMove**, the table for the new move is looked up. In previous programs, the index was always zeroed to start the table at the beginning. Here it is sometimes different. All movement tables finish to smoothly transition into **Fwd** so the offset is zeroed then. We are assuming where the legs are when checking the bumpers so **BumpL** and **BumpR** start at zero too. This is because the tables took into account where the legs would be when they were made. When Toddler "sees" something, the walking sequence can be anywhere in the **Fwd** table. To prevent tripping over its own feet, Toddler will take the offset being used in the **Fwd** table and use it as a starting point into the **PivotLeft** or **PivotRight** table. Since **Fwd** is duplicated at the front of both Pivot tables, there is a smooth transition. This section is worth further study when you want to add other moves or other conditions for table transitions.

## FSM

**New\_Movement** is similar to **Do\_Movement** in that it is also entered with a new state desired (**newMove**) and it had its current state (**currentMove**) saved. This style of subroutine is called a Finite State Machine or FSM. Remember back to Activity #4, Approach 3, where state diagrams were used to analyze the stepping? State diagrams will work here too. The state diagram for this code is pretty simple so it won't be diagrammed but you get the idea. FSMs are excellent for real time programming because they are a convenient way to be able to call a subroutine, have it do what it has to, and quickly get back out. Diagramming the states can be very helpful in understanding more complex FSMs but these are all pretty simple so there is no need to diagram them.

### Commented out DEBUGs

The `DEBUG` statements with the comment (`*`) symbol in front don't get executed or take up program memory space. Why bother? Well, when trying to understand the program, or trying to debug your modifications to it, `DEBUG` statements are very helpful. By doing a global replace of `*DEBUG` with `DEBUG`, you can get a screen on the Debug Terminal of your PC that will show what is happening in the program. Using the BASIC Stamp Editor's Debug Terminal means the serial cable has to stay connected to the Toddler while the program is running. The `DEBUG` delays will also cause the program to completely lose any possibility of meeting the 20 ms servo timing. Walking isn't too effective with the cable sticking out the back and the extended servo timing but you can hold Toddler in the air and watch what it will try to do in response to the different sensor inputs. Or, simply switch the switch to position 1 so the servos are not activated.

### State Indicator LEDs

The four LEDs connected to P5-P8 show what mode Toddler is in without needing a cable connected or slowing the timing to the servos. They are also fun to watch.

### Do\_Vision

Let's now skip back up to the first subroutine. `Do_Vision` checks the IR sensors like the earlier programs. This takes a few milliseconds. If it already saw something from before, the `leftIR_Flag` and/or the `rightIR_Flag` will be set. If a flag is set, the vision movement continues with the previous decision on movement. This allows the full sequence to be run from a momentary incident of "seeing" something. If neither flag was set, a direction decision is made based on the current state of the IR sensors. Wait a minute! Earlier it was flatly stated not to do things that didn't need to be done in the real time loop. If either the `leftIR_Flag` or the `rightIR_Flag` is set, the code doesn't even look at the result of the sensing. Why keep it in the high-speed loop?

Second rule of real time programming - Sometimes do it in the high-speed loop, even if you didn't have to?

Remember the goal here isn't to get the code to run the fastest. It is to get the program to consistently output servo pulses at 20 ms intervals. If, while running the `PiVR` and `PiVL` movement tables, the `FREQOUT` commands were skipped, the servo pulses would come at a 2 ms faster rate. In this case the `FREQOUT` commands are really just equivalent to a `PAUSE` used to even the times between servo pulses.

## Do\_Bumper

Like `Do_vision`, this is another FSM that tests sensors and flags and decides where to go next. An important thing to notice though, `Do_Bumper` outputs its decision into the same variable (`newMove`) that `Do_vision` does. It will simply overwrite whatever `Do_vision` decided. Why?

## Subsumption Architecture

One way to program a robot is to have the program look at all possible internal states and inputs and then decide on a course of action from that. The disadvantage to that style is that it can get difficult to add new behaviors or sensors without redoing almost the whole program.

Another style of program, called “Subsumption Architecture” or “Behavior Based Programming”, is where the robotic control is broken into separate behaviors. In our case we have two, vision and bumper. Each behavior (subroutine) decides what the output (`newMove`) should be based on its inputs and its internal state. A higher priority behavior can “subsume” or take control from a lower priority behavior. In our case, `Do_Bumper` subsumes `Do_vision` by writing over `newMove`. Notice `Do_Bumper` doesn’t always write `newMove`, only when something is happening with the bumpers. Otherwise it leaves `newMove` alone. No bumper activity therefore causes `Do_vision` to have control. This lets the behaviors have more independent control over their domain without making the overall program too complicated. Normally, a behavior will keep track of what it wants the output to do and for how long and that is what happens if it is not subsumed. When whatever was subsuming it goes away, it will immediately take control. This gives rise to “emergent behaviors”. Those are behaviors that weren’t explicitly programmed in. They just happen out of the interactions of the sensor inputs and behaviors. In our case, however, that would be dangerous. If we are just done the bumper movement, we can’t just start in the middle of a vision movement that had been subsumed by the bumper. To prevent that in our case, `clr_vision` resets the vision LEDs and status bits if the bumper flags are active. This type of code is normally not used with Subsumption Architecture and would not be used on a wheeled type robot.

Our program has only one output, a single movement so either programming style will work here. Behavior Based Programming will be more helpful to you when you are writing programs that have many outputs and many sensors. To add a behavior to this program, insert the behavior subroutine call into `Main_Program`. Any behavior calls

before the added one have lower priority and any after it have higher priority. Professor Rodney Brooks of MIT is widely credited with first publishing on Subsumption Architecture and there is lots of information on the web if you want to study it further.

### Final thoughts

Now we have a robot that will actively look at its sensors while it is moving. It can even make decisions about what to do in real time. One warning though, notice the constant `MoveDelay`. `Main_Program` uses it in a `PAUSE` statement to make up the difference between the time taken to run one loop and the 20 ms needed by the servo. The delay value here is only 1 millisecond! In other words the code needed 19 of the 20 ms allowed to do its job. In real time programming, you only have to just get there so we are fine. Actually, if we had a very firm 20 ms specification, we would fail because when we set up new movement tables, we go beyond the 20 ms. That doesn't happen often, and the servos do forgive it, so it's fine. Problems might come up later though as you start to add features or options to the program. After all, there's still two-thirds of the program space and almost one-half of the variable space left.

## CHALLENGES

- √ In the example program, once Toddler hit something, it continued with the backup and turn sequence without looking if it was hit again. How can you fix that to include still watching the bumpers while actually doing the **Rbumper** or **LBumper** pattern? Hint: How did we do it when the bumper was subsuming the vision?
- √ In Program 8.3 the bumpers detect an object and Toddler takes one slide turn. Modify the source code so when the bumpers are activated the Toddler takes three or four turns away from the object it detected.
- √ Toddler Toes have bi-color LEDs on them. Rewrite Program 8.3 so that the LEDs are red while the Toddler moves forward and green when it is backing up. Hint: Neither red or green LED should be on when you are checking the bumper switches.
- √ In the program, Toddler assumed that if it hit something, it was with the leg going forward at the time. Can you devise a way to sense if the bumper was hit when its leg wasn't moving forward? As an example, maybe your cat walked up to toddler and touched the bumper on the foot that was on the ground at the time. Hint: the difficult part here can be deciding what you want Toddler to do when it happens.



## Chapter #9: Acceleration-Based Motion

*Note: This chapter requires the Memsic 2125 Accelerometer (#28017) from Parallax.*

### MEASURING TODDLER'S INCLINE

The Memsic 2125 is a great little accelerometer. You can add it to your Toddler to actually sense and control how far it is tilted over rather than using the current method of open-loop control.

The Memsic 2125 is a dual-axis accelerometer. Inside, the Memsic 2125 has a small heater that warms a "bubble" of air. When the sensor moves, this heat bubble moves and is detected by thermopiles that surround the heater. Changes detected by the thermopiles are conditioned and the outputs are pulses (one for each axis) that correspond to the forces acting on the sensor.

The accelerometer comes in a surface-mount package so Parallax has made a DIP carrier that is more convenient for use on your Toddler. The tilt output is a 100 Hz square wave (10 ms period) whose duty cycle depends on tilt. 50% duty cycle (high for half the time, ground for the other half) means the sensor is level in the direction being read.

**Applications for the Memsic 2125** are located throughout the Parallax web site. With the Toddler the sensor is only utilized for single-axis incline measurement, but it can also measure angular rotation and acceleration on two axis. Here is a sampling of the material available for download from the web:

- "It's All About Angles" in the [Nuts and Volts of Stamp Applications](#) column #92. This article demonstrates the BASIC Stamp integer math for calculating g-force and inclination ([http://www.parallax.com/detail.asp?product\\_id=28168](http://www.parallax.com/detail.asp?product_id=28168)).
- "Memsic 2125 Demo Kit" documentation ships with the purchase of the Memsic 2125 accelerometer from Parallax and demonstrates the angular measurement examples ([http://www.parallax.com/detail.asp?product\\_id=28017](http://www.parallax.com/detail.asp?product_id=28017)).
- Memsic 2125 datasheet (<http://www.memsic.com>)
- Prior to using the accelerometer on your Toddler, we recommend that you download and run some of the sample BASIC Stamp 2 programs from the above links. These articles contain very instructional background code which is not repeated in this text.



When you are tilting the sensor, one way is positive G, the other way is negative G. The sensor has a range of  $\pm 2g$ .

## MORE ON REAL TIME TECHNIQUES

Since we plan to use the sensing in the high speed real time servo loop, can we make **Read\_Y\_Force** faster? Sure! Change the 10 to 5 and you don't have to multiply **incline** by 2 before using it in the equation. Why should the BASIC Stamp have to do it millions of times if you can do it once?

```
Read_Y_Force:
  PULSIN Yout, 1,incline
  mGForce = ((incline / 5) - 500) * 8
```

Remember the resolution of **PULSIN** is 2us. If the first thing we do is divide it by 5, we end up with an equivalent resolution of only 10 us. Why waste our resolution?

$$\text{MGForce} = ((\text{incline} / 5) - 500) * 8 = ((8 * \text{incline}) / 5) - (500 * 8) = ((8 * \text{incline}) / 5) - 4000$$

Now we multiplied by 8 before dividing so we'll keep the maximum resolution possible. You'll have to make sure there can't be an overflow if we multiply (left shift) first. Since the period is 10 ms, **incline** can never be over 10,000/2 or 5000. Since  $8 * 5000 = 40,000$  we can't overflow a word variable. After dividing by 5, we get back down to 8000. That won't overflow a signed word variable at roughly  $\pm 32000$ .

```
Read_Y_Force:
  PULSIN Yout, 1,incline
  mGForce = ((incline * 8) / 5) - 4000
```

Multiplies take a long time. Eight is an exact binary number. All we have to do is left shift the word variable **incline** 3 times and we have multiplied by 8.

```
Read_Y_Force:
  PULSIN Yout, 1,incline
  mGForce = ((incline / 5) - 500) <<3

Read_Y_Force:
  PULSIN Yout, 1,incline
  mGForce = ((incline <<3) / 5) - 4000
```

There, much faster and with better resolution too.

But after all this, what will we do with the G force anyway? Aren't we just going to compare the pulse width output to a limit for each direction? In that case, why bother to

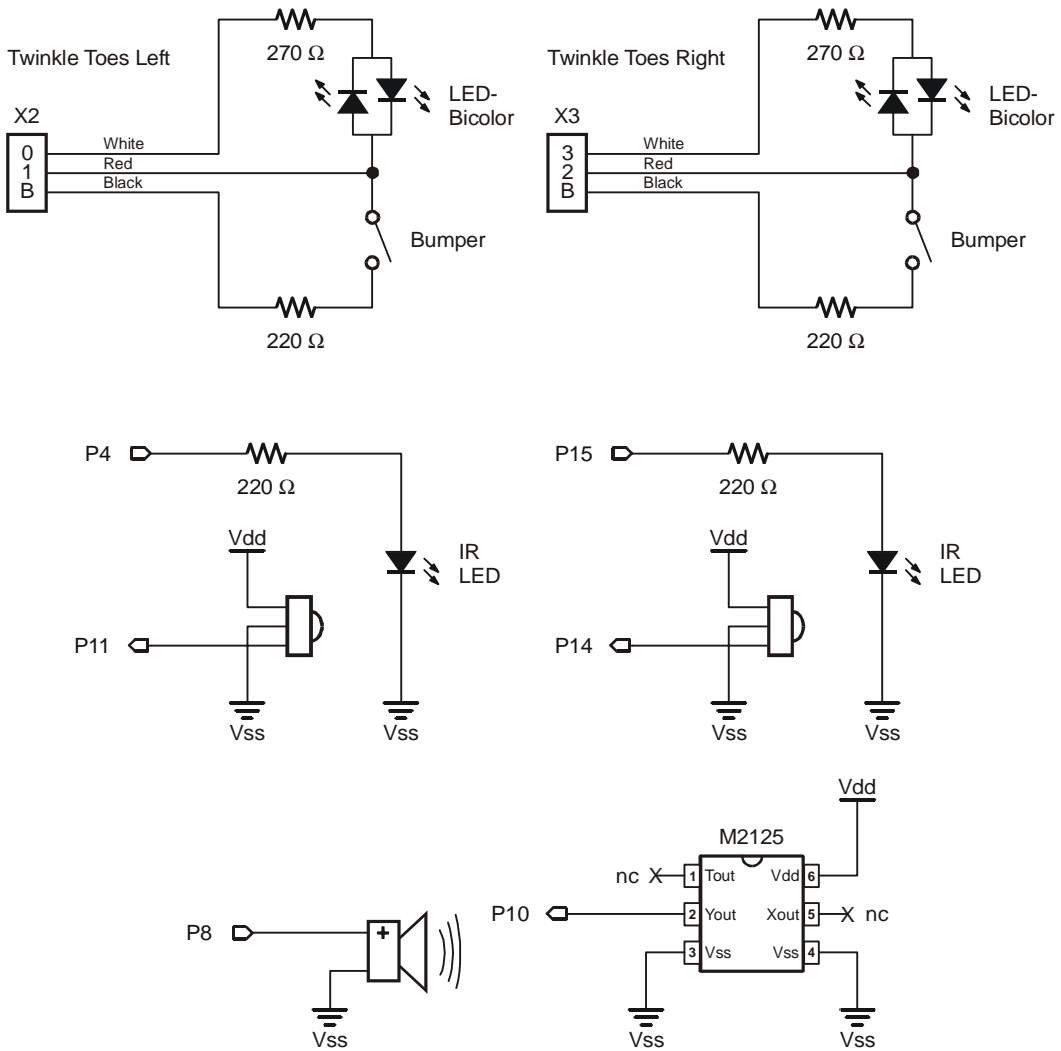
calculate anything? Just set the comparison to a limit value for incline and don't bother to calculate anything! – Remember the First rule of real time programming - If you don't absolutely have to do it in the high-speed loop, don't!

Let's get into it now.

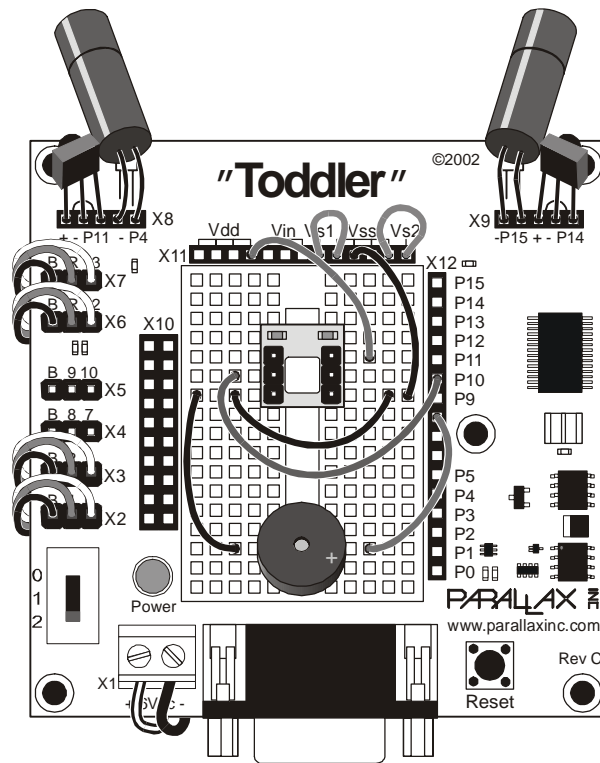
### **ACTIVITY #1: BUILDING AND TESTING THE ACCELEROMETER**

The circuit is exactly the same as that in Chapter 8 except for the addition of the accelerometer.

Figure 9-1 is the circuit schematic and Figure 9-2 is a pictorial showing proper mounting of the Memsic 2125. The pictorial shows two important points about the accelerometer – placement on the Toddler board and the location on pin 1 for wiring purposes. Add the accelerometer to the prototyping area of your Toddler board. Make sure when putting the sensor down that it is flat on the board and correctly facing forward.



**Figure 9-1:** Add the Memsic 2125 accelerometer to your current Toddler circuitry. Additionally, you can add two LED indicators on P5 and P6 to show the status of the infrared detectors.



**Figure 9-2:** Pictorial of Memsic 2125 Circuit. X2 and X3 are connected to the Twinkle Toes. X6 and X7 are connected to tilt and stride servos. Additionally, you may connect two LEDs to P5 and P6 for visual feedback of the infrared detectors.

Once the hardware is wired, enter and run Program Listing 9.1

```

' -----[ Title ]-----
' Toddler Program 9.1: Memsic Accelerometer Test Program
' Incline feedback determines tilt limitations of Toddler
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----
Yout          PIN    10          ' Yout from Memsic 2125
TiltServo     PIN    13          ' Toddler tilt servo
StrideServo   PIN    12          ' Toddler stride servo

' -----[ Constants ]-----
TiltCenter    CON    750          ' TiltServo center position
StrideCenter  CON    750          ' StrideServo center position
StepVal       CON    2            ' Servo step value = 1 or 2
' Set StepVal = 3 if using 'DEBUGs
escVal        CON    4            ' Indicator for excessive tilt
Delay         CON    8            ' Pause between servo pulses -
' add 8 ms to get 20 total
RightLimit    CON    500          ' Limit does not occur on flat
LeftLimit     CON    1000         ' Limit does not occur on flat
' Right and left depend on
' how you mounted the Memsic
' sensor. Below assume pins 1 & 6
' are toward front of Toddler.
YAxisRtLimit  CON    2600         ' Yout right limit *2us = 5.34 ms
YAxisLfLimit  CON    2400         ' Yout left limit *2us = 4.500 ms

' -----[ Variables ]-----
Incline       VAR    Word         ' Pulses from Memsic 2125
ServoLoop     VAR    Word         ' General purpose loop var
Counter       VAR    Nib          ' Counter filters acceleration
tiltCount     VAR    Nib          ' Temporary counter

' -----[ Initialize ]-----
ServoLoop = 750

' -----[ Main Code ]-----
RightFromCenter:
  Counter = 0
  FOR ServoLoop = ServoLoop to RightLimit STEP StepVal
    PULSOUT TiltServo, ServoLoop
    PULSOUT StrideServo, StrideCenter

```

```

PAUSE Delay
PULSIN Yout,1,Incline
'DEBUG "Right from Center - ServoLoop = ", DEC ServoLoop
'DEBUG ": Incline = ", DEC Incline, CR
IF (Incline > YAxisRtLimit) THEN Counter = Counter + 1
'DEBUG "Counter = ", DEC Counter
'DEBUG ": tiltCount = ", DEC tiltCount, CR
IF Counter > escVal THEN LeftFromRight
NEXT
'DEBUG "ServoLoop = ", DEC ServoLoop,": Incline = ", DEC Incline, cr

LeftFromRight:
Counter = 0
FOR ServoLoop = ServoLoop to LeftLimit STEP StepVal
PULSOUT TiltServo, ServoLoop
PULSOUT StrideServo, StrideCenter
PAUSE Delay
PULSIN Yout,1,Incline
'DEBUG "Left from Right - ServoLoop = ", DEC ServoLoop
'DEBUG ": Incline = ", DEC Incline, CR
IF (Incline < YAxisLfLimit) THEN Counter = Counter + 1
'DEBUG "Counter = ", DEC Counter,": tiltCounter = ", DEC tiltCount, cr
IF Counter > escVal THEN RightFromLeft
NEXT
'DEBUG "ServoLoop = ", DEC ServoLoop,": Incline = ", DEC Incline, cr

RightFromLeft:
Counter = 0
FOR ServoLoop = ServoLoop to RightLimit STEP StepVal
PULSOUT TiltServo, ServoLoop
PULSOUT StrideServo, StrideCenter
PAUSE Delay
PULSIN Yout,1,Incline
'DEBUG "Right from Left - ServoLoop = ", DEC ServoLoop
'DEBUG ": Incline = ", DEC Incline, CR
IF (Incline > YAxisRtLimit) THEN Counter = Counter + 1
'DEBUG "Counter = ", DEC Counter,": tiltCount = ", DEC tiltCount, cr
IF Counter > escVal THEN LeftFromRight
NEXT
'DEBUG "ServoLoop = ", DEC ServoLoop,": Incline = ", DEC Incline, CR

GOTO LeftFromRight
END

```

There are two ways to run program 9.1.

The first way is as it is printed above. Toddler will tilt back and forth going between the `YaxisRtLimit` and `YaxisLfLimit` values. Be sure to support your Toddler while it is moving as it may fall over.

The second way is to globally replace ``DEBUG` with `DEBUG` first. Toddler will go much slower because the servo refresh rate is so slow but Toddler will indicate the measured tilt (actually just the pulse width/2) values on the `DEBUG` screen on your PC.

Watch the values as Toddler goes from side to side. Pick the robot up and tilt left and right it to get a feel for how the Memsic sensor works and what values correspond to what tilt.

### **Noise Filter**

The counter in the Program 9.1 allows filtering the tilt values. If it weren't there, the tilting would stop the first time a value over the limit was read. The sensor is looking at G forces. Anything touching the Toddler while it is sensing could cause an out of limit value to be read. The `IF` statements and counter require multiple measurements outside the limit before acting.

## **ACTIVITY #2: INTEGRATED SENSOR WALKING PROGRAM**

Enter and run Toddler Program 9.2.



```

' -----[ Title ]-----
' Toddler Program 9.2 Complete Sensor Management.bs2
' This program has real time servo programming, inclinometer sensing
' Toddler bumper toes detection and infrared object detection
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

TiltServo      PIN    13      ' tilt servo on X7
StrideServo    PIN    12      ' stride servo on X6

LeftIRLED      PIN    4        ' left IRLED
RightIRLED     PIN    15       ' right IRLED
LeftIR_Sensor  PIN    11       ' left IR sensor input
RightIR_Sensor PIN    14       ' right IR sensor input

Spkr           PIN    8        ' beeper pin
Yout           PIN    10       ' Yout from Memsic 2125

LeftBumper     PIN    2        ' bumper switch on left foot
RightBumper    PIN    1        ' bumper switch on right foot

LeftBumpPU     PIN    3        ' left bumper pull up
RightBumpPU    PIN    0        ' right bumper pull up

' Indicator LEDs are optional, but useful for seeing movements

LeftVisionLED  PIN    5        ' left vision display LED -
RightVisionLED PIN    6        ' right vision display LED -

' -----[ Constants ]-----

MoveDelay      CON    0        ' in milliseconds - normally 20
' code is already using 30 ms

TiltStep       CON    5        ' experiment with this value

RightTilt      CON    500      ' tilt limits was 620
CenterTilt     CON    750
LeftTilt       CON    1000     ' was 880

StrideStep     CON    7        ' StrideServo step size

RightStride    CON    625      ' stride limits was 650
CenterStride   CON    750
LeftStride     CON    875      ' was 850

ctrlLEDs      CON    %1111    ' define DIRs as constants to
prepBumper    CON    %1001    ' simplify LED/bumper setups

```

```

bothGreen      CON      %0110      ' OUTa register sets red/green
bothRed        CON      %1001      ' status on P0-P3 for Twinkle Toes
rightRed       CON      %0101      ' object detection states
leftRed        CON      %1010
bothLEDsOff    CON      %1001

Fwd            CON      0          ' codes to pick movement table
PivL           CON      1          ' set newMove to one of these
PivR           CON      2          ' values and New_Movement will LOOKUP
BumpL          CON      3          ' the right movement table
BumpR          CON      4

escVal         CON      4          ' Indicator for excessive tilt

' Right and left depend on
' how you mounted the Memsic
' sensor. Below assume pins 1 & 6
' are toward front of Toddler.
' Use your values here
YAxisRtLimit   CON      2560      ' Right limit - was 2570 = 5.14 ms
YAxisLfLimit   CON      2460      ' left limit - was 2350 = 4.70 ms
' Yout center - is 2510 = 5.02 ms

' -----[ Variables ]-----

' Bit variable flags set so 1 causes the action to be taken
doneTiltFlag   VAR      Bit        ' Tilt servo reached new value
doneMoveFlag   VAR      Bit        ' Servos have reached new values
rightIR_Flag   VAR      Bit        ' See something on right side
leftIR_Flag    VAR      Bit        ' See something left side
rightBumpFlag  VAR      Bit        ' Bumper hit on right side
leftBumpFlag   VAR      Bit        ' Bumper hit on left side

sensors        VAR      Nib        ' Lower 2 bits of the sensors
' variable used to store IR values

currentMove    VAR      Nib
newMove        VAR      Nib

Counter        VAR      Nib        ' Filters tilt sensor output

Mx             VAR      Word       ' index for movement tables
MxOffset       VAR      Byte       ' added to Mx for index
bMovmnt        VAR      Byte       ' table value for movement lookup

currentTilt    VAR      Word
currentStride  VAR      Word
newTilt        VAR      Word
newStride      VAR      Word

Incline        VAR      Word       ' Pulses from Memsic 2125

```

```

' -----[ EEPROM Data ]-----
'
' These are actual values saved in the Basic Movement tables.

TL          CON      0          ' use lower nibble for these
TC          CON      1
TR          CON      2

SL          CON      3
SC          CON      4
SR          CON      5

xx          CON      255        ' table end code

' ----- Basic Movement Tables -----
'
' These tables contain Basic Movements consisting of bytes containing
' above Basic Movement Codes to describe movement sequences.
' An xx indicates the end of a list.
' Note that PivotLeft and PivotRight aren't necessarily entered at the
' of their tables

Forward     DATA    TR, SL, TL, SR, xx
PivotLeft   DATA    TR, SL, TL, SR
            DATA    TC, SL, TL, SR, xx
PivotRight  DATA    TR, SL, TL, SR
            DATA    TR, SL, TC, SR, xx
RBumper     DATA    SL, TR, SR, TC, SL, TL, SR, xx
LBumper     DATA    SR, TL, SL, TC, SR, TR, SL, TL, SR, xx

' -----[ Initialize ]-----

DIRS = ctrlLEds
OUTS = bothGreen
GOSUB Clr_Vision          ' reset vision LEDs and flags
Counter = 0              ' reset tilt filter

ResetFeet:
  newTilt = CenterTilt
  newStride = CenterStride
  currentTilt = CenterTilt
  currentStride = CenterStride

  FOR currentMove = 1 TO 15          ' center feet for .3 sec
    GOSUB Move_It                  ' currentMove is only a nibble
    PAUSE 15                       ' make up 20 ms servo refresh time
  NEXT

'DEBUG "Forward = ", DEC Forward, cr
'DEBUG "PivotLeft = ", DEC PivotLeft, CR, "PivotRight = "

```

```

'DEBUG DEC PivotRight,CR
'DEBUG "RBumper = ", DEC RBumper, CR, "LBumper = ", DEC LBumper, CR

doneTiltFlag = 1
doneMoveFlag = 1
sensors = 0
MxOffset = 0

FREQOUT Spkr, 2000, 3000      ' program start/restart signal.
rightBumpFlag = 0
leftBumpFlag = 0

currentMove = 15             ' invalid value to assures start
' newMove = Fwd              ' for testing single moves -
' newMove = PivL             ' comment out GOSUBs to vision
' newMove = PivR            ' and bump or the value might be
' newMove = BumpL           ' overwritten
' newMove = BumpR

' -----[ Main Code ]-----

Main_Program:
DO
  GOSUB Do_Vision             ' look for obstacles with IR
  GOSUB Do_Bumper            ' check for obstacles with bumpers
  GOSUB New_Movement         ' generates next move from table
  GOSUB Do_Movement          ' gives servo pulses out
  'PAUSE MoveDelay           ' no pause required because the
                              ' PBASIC code is using 30 ms

LOOP
END

' -----[ Subroutines ]-----

Do_Vision:

FREQOUT LeftIRLED,1,38500    ' pulse left IRLED.
sensors.BIT0 = LeftIR_Sensor ' store IR detector output in RAM.

FREQOUT RightIRLED,1,38500   ' repeat for the right IR pair.
sensors.BIT1 = RightIR_Sensor

                              ' check if currently doing IR move
IF ((leftIR_Flag = 1) & (rightIR_Flag = 1)) THEN See_Both
IF (leftIR_Flag = 1) THEN See_Left  ' is left only?
IF (rightIR_Flag = 1) THEN See_Right ' is right only?

' Load IR detector output values into the lower 2 bits of the sensors
' variable, storing a number between 0 and 3 that the BRANCH command
' can execute the appropriate routine.

```

```

BRANCH sensors, [See_Both, See_Right, See_Left, See_None]

See_Both:
  newMove = PivR
  rightIR_Flag = 1           ' flag: 1 = something on right side
  leftIR_Flag = 1           ' flag: 1 = see something left side
  HIGH LeftVisionLED       ' left vision sequence display LED
  HIGH RightVisionLED      ' right vision sequence display LED
  RETURN

See_Right:
  newMove = PivL
  rightIR_Flag = 1         ' flag: 1 = something on right side
  HIGH RightVisionLED     ' right vision sequence display LED
  RETURN

See_Left:
  newMove = PivR
  leftIR_Flag = 1         ' flag: 1 = something left side
  HIGH LeftVisionLED     ' left vision sequence display LED
  RETURN

See_None:
  newMove = Fwd
  RETURN

'-----

Do_Bumper:
  DIRS = prepBumper
  OUTS = bothLEDsOff
  ' Bumpers on each foot. A hit causes Toddler to back and turn.
  ' Code assumes leg that hit is on the leg that is moving
  ' (while leaning on the other leg).
  ' A hit backs up the hit leg right away then steps back 1 step
  ' and does a pivot away.

  IF (leftBumpFlag = 1) THEN Bump_LeftA
  IF (rightBumpFlag = 1) THEN Bump_RightA

  ' check for old bumper hit on left
  ' check for old bumper hit on right

  IF (LeftBumper = 0) THEN Bump_Left
  IF (RightBumper = 0) THEN Bump_Right
  DIRs = ctrlLEDs
  OUTS = bothGreen

  RETURN

```

```

Bump_Left:
  'DEBUG "bump left", CR          ' sends only once on change

Bump_LeftA:
  leftBumpFlag = 1              ' flag indicates bump response move
  newMove = BumpL
  DIRS = ctrlLEDs
  OUTS = leftRed
  GOTO Clr_Vision                ' goes to Clr_Vision

Bump_Right:
  'DEBUG "bump right", CR

Bump_RightA:
  rightBumpFlag = 1
  newMove = BumpR
  DIRS = ctrlLEDs
  OUTS = rightRed
                                ' falls through to Clr_Vision
'-----

Clr_Vision:
                                ' reset vision LEDs and flags

  rightIR_Flag = 0              ' flag: 1 = something on right side
  leftIR_Flag = 0               ' flag: 1 = something left side
  LOW  LeftVisionLED            ' left vision sequence display LED
  LOW  RightVisionLED           ' right vision sequence display LED
  RETURN

'-----

New_Movemt:
                                ' sequence for forward motion

  IF (newMove <> currentMove) THEN ' if new movement visit DATA table

    'DEBUG CR, "start NewMovemt; newMove = ", DEC newMove
    'DEBUG ", currentMove = ", DEC currentMove, CR

    doneMoveFlag = 1            ' stop current move wherever it is
    currentMove = newMove

    ' When customizing the program, here is where to integrate between
    ' moves. You can make the transitions reasonable by using the existing
    ' Mx value and MxOffset to compute the proper MxOffset to start into
    ' the next move.

    IF ((newMove = Fwd) OR (newMove = BumpL) OR (newMove = BumpR)) THEN
      MxOffset = 0              ' reset MxOffset value for new move
    ENDIF

    ' note: kept old MxOffset value for lead into pivot left or right

```

```

LOOKUP newMove, [Forward, PivotLeft, PivotRight, LBumper, RBumper], Mx
' lookup movement table index
ENDIF

IF (doneMoveFlag = 1) THEN
Counter = 0
READ (Mx + MxOffset), bMovmnt
'DEBUG "Mx = ", DEC Mx, ", MxOffset = ", DEC MxOffset
MxOffset = MxOffset + 1

IF (bMovmnt < xx) THEN
'DEBUG " bMovmnt = ", DEC bMovmnt, " "

SELECT bMovmnt
CASE TL : newTilt = LeftTilt : 'DEBUG "TL, ", CR
CASE TC : newTilt = CenterTilt : 'DEBUG "TC, ", CR
CASE TR : newTilt = RightTilt : 'DEBUG "TR, ", CR
CASE SL : newStride = LeftStride : 'DEBUG "SL, ", CR
CASE SC : newStride = CenterStride : 'DEBUG "SC, ", CR
CASE SR : newStride = RightStride : 'DEBUG "SR, ", CR
ENDSELECT
' will fall through if invalid index
ELSE
'DEBUG " xx = finished table", CR, CR
MxOffset = 0
leftBumpFlag = 0
rightBumpFlag = 0
DIRS = ctrlLEds
OUTS = bothGreen
GOSUB Clr_Vision
GOSUB Do_Vision
GOSUB Do_Bumper
GOTO New_Movemt
ENDIF

ENDIF

RETURN
' ignore if invalid

'-----
Do_Movement:

doneMoveFlag = 0
doneTiltFlag = 0

IF (currentTilt = newTilt) THEN
doneTiltFlag = 1
Counter = 0
PAUSE 1
ELSE

```

```

IF (currentTilt < newTilt) THEN      ' start tilt calculations
  currentTilt = currentTilt + TiltStep
                                     ' from right tilt = increment
  PULSIN Yout,1,Incline
  'DEBUG "Incline = ", DEC Incline, CR
  IF Incline < YAxisLfLimit THEN
    Counter = Counter + 1
    'DEBUG "Counter = ", DEC Counter, CR
  ENDIF
  IF Counter > escVal THEN
    newTilt = currentTilt              ' at limit of tilt, move no more
  ENDIF
  IF currentTilt >= newTilt THEN
    currentTilt = newTilt              ' correct if overshoot tilt
    doneTiltFlag = 1
    Counter = 0
  ENDIF
ELSE
  currentTilt = currentTilt - TiltStep
                                     ' from left tilt = decrement
  PULSIN Yout,1,Incline
  'DEBUG "Incline = ", DEC Incline, CR
  IF Incline > YAxisRtLimit THEN
    Counter = Counter + 1
    'DEBUG "Counter = ", DEC Counter, CR
  ENDIF
  IF Counter > escVal THEN newTilt = currentTilt
  IF currentTilt <= newTilt THEN
    currentTilt = newTilt              ' stop at limit of tilt
    doneTiltFlag = 1
    Counter = 0
  ENDIF
ENDIF
ENDIF
ENDIF

Move_Stride:

IF (currentStride < newStride) THEN
  currentStride = currentStride + StrideStep
                                     ' from right stride = increment
  IF (currentStride >= newStride) THEN
    currentStride = newStride
    IF (doneTiltFlag = 1) THEN doneMoveFlag = 1
                                     ' uses doneMoveFlag to increment
                                     ' states
  ENDIF
ELSE
  currentStride = currentStride - StrideStep
                                     ' from left stride = decrement
  IF (currentStride <= newStride) THEN' jump if still moving
    currentStride = newStride

```



```

        IF (doneTiltFlag = 1) THEN doneMoveFlag = 1
        ENDIF                                ' uses doneMoveFlag to increment
        ENDIF                                ' states
Move_It:
    PULSOUT TiltServo,    currentTilt
    PULSOUT StrideServo, currentStride
    RETURN

```

**Do\_Movement** is the big change here. The tilt section kept the limits for the servo pulse width but opened the values out to their full range. Measuring the tilt sensor was integrated with the tilt movement.

If you recall, the program from the last experiment just managed to get the servo pulses out at 20 ms intervals. Now we just added a measurement that takes around 5 ms and some more calculations. When the tilt is ramping, the servo refresh is 30 ms. Not as good as it can be! A quick and dirty fix was to check up front if the tilt is already where it is supposed to be. If it is, we set the flag and skip the tilt sensing altogether. Since that would run at 19 ms servo refresh, a 1 ms **PAUSE** was added. When we have to do the tilt measurements, the refresh time goes back to 30 ms. It works like that but any changes to this program could cause the servo movements to get rough. As it is, the same step values for tilt are slower for tilting.

9

### **Where's the Problem?**

The whole issue is the need to keep pulsing the servos every 20 ms. Toddler really doesn't need to use its vision and check the bumpers every 20 ms. You could set up the state machines for vision and bumpers to run every other time. On one pass through the main **DO...LOOP**, do the IR, the next the bumpers. That trick will minimize the maximum time between servo pulses and get smoother performance back. The big question now becomes – “How do I measure the servo pulse cycle times?” Actually, if you have an oscilloscope, it is easy. But many people don't have one so that brings us back to the problem.

### **Co-processor: Another Solution**

Tuning a processor to get the absolute maximum performance out of it is an interesting and challenging exercise. It can also be a big pain and require lots of time spent with an

oscilloscope trying to find a quicker way of executing your code. If your goal is to try out new and fun behaviors, or sensors, or whatever, on your Toddler, there is another way. Since the whole problem was the servos and their need for regular updates you can use a “Co-Processor” that is specially programmed to handle the servos. There are two major distinctions in “servo controller” Co-Processors. One type is given the new servo position and it sends the servo to that position at full speed. It then gives the same pulse widths every 20 ms to the servo until you send a new value. That type is better suited for wheeled robots with modified servo motors. They aren’t very helpful in a walking robot. The issue with a walker is that it is constantly changing the value of the servo output pulse in a ramped fashion. As you recall, that is what the `StrideStep` and `TiltStep` were used for in the `Do_Movement` subroutine. Happily, there is another type. A “ramping” servo controller remembers where each servo is and ramps (adds constant steps like `Do_Movement` did) to get to the new position. It essentially executes the `Do_Movement` subroutine for you. Many Co-Processors have more than just servo controllers in them. One we recommended is from Blue Bell Design Inc. ([www.bluebelldesign.com](http://www.bluebelldesign.com)). It has 8 ramping servo controllers, 5 analog to digital converter inputs (10-bit), and extra inputs and outputs. Best of all, it will be available on a compatibly sized PC board for easy installation on your Toddler! The Blue Bell Design Co-Processor is discussed in Appendix C.

## CHALLENGES

- √ In `Do_Movement`, the `PULSIN` command reads the value of the Memsic 2125 accelerometer. Since we know the code is too slow to keep up with 20 ms refresh on the servos, and there is plenty of code space left, is there some way you can use the current state of the accelerometer signal to pick up an average of 1.5 ms in the worst case? What is the worst case state of `yout`? Why is it the worst case? Hint: What do you know about how `PULSIN` works that makes a difference?
- √ Tune the timing of your Toddler for optimal speeds, tilt, and stride. How good can you get it to be?
- √ You know that Toddler really doesn’t need to use its vision and check the bumpers every 20 ms. Can you think of ways to make the state machines in vision and bumpers run on alternate cycles? What happens to the evenness of the servo steps if the duration between servo pulses changes cycle to cycle?.

## Appendix A: Parts Listing and Sources

All parts used in the Toddler kit are available individually from the Parallax Component Shop at [www.parallax.com](http://www.parallax.com). If you can't readily find the component you are looking for in the Component Shop enter the name of it in the on-line search box using the stock code.



The Toddler Kit comes in two flavors:

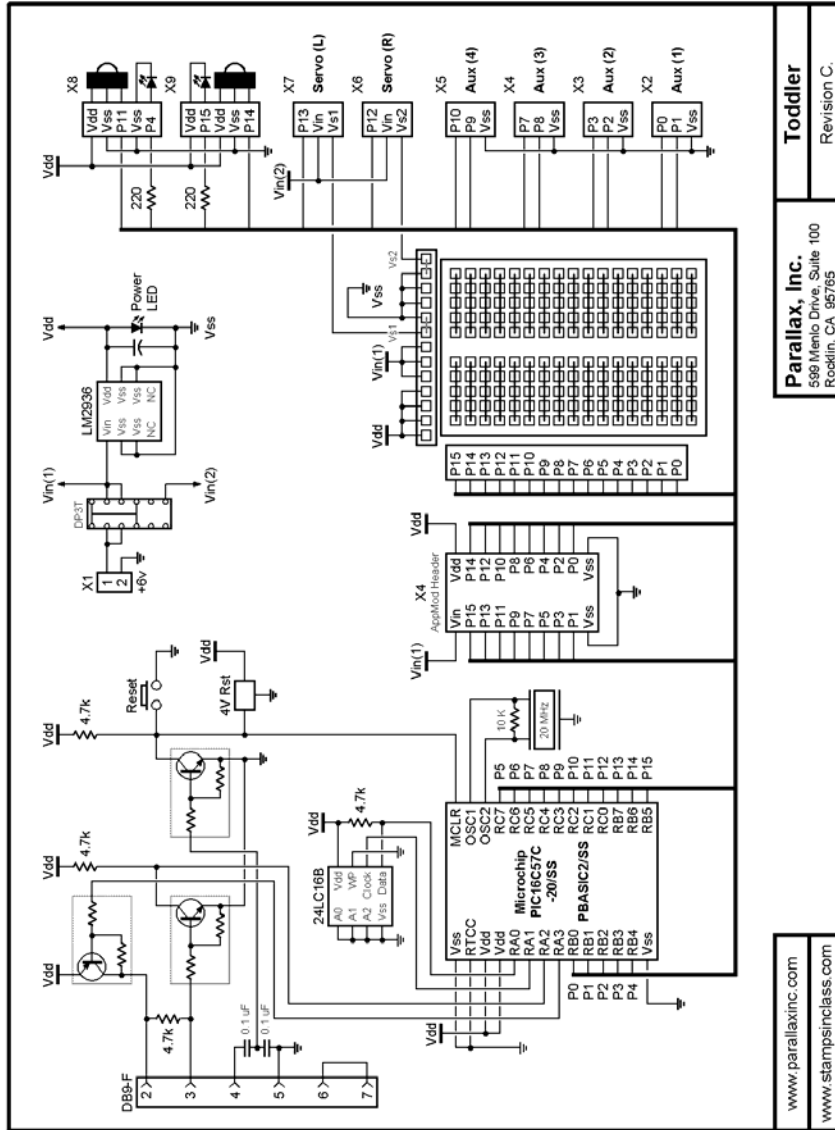
- Gold Anodized Toddler Kit - 27310
- Blue Anodized Toddler Kit - 27311

<b>Toddler Kit Bill of Materials</b>		
Parts and quantities subject to change without notice		
Parallax Part #	Description	Qty/Kit
<b>Electronic Components</b>		
150-02210	220 Ohm resistors ¼ watt 5% tolerance	4
200-01040	0.1 uF capacitor	2
200-01031	0.01 uF capacitor	2
350-00001	Green LED	2
350-00003	Infrared LEDs	4
350-00007	Yellow LED	2
350-00009	Photoresistors	2
350-00014	Infrared detector	4
350-90000	LED standoffs	4
350-90001	LED light shields	4
550-00020	Toddler Printed Circuit Board with BASIC Stamp 2	1
753-00001	Battery Pack with Tinned Wires	1
800-00016	Jumper wires (bag of 10)	1
900-00001	Piezospeaker	1
900-00010	Parallax Toddler Servo (Toddler Mini F BB)	2
<b>Metal Parts</b>		
720-00001	Toddler Top Plate - Gold Anodized	1
720-00002	Toddler Top Plate - Blue Anodized	1
720-00003	Toddler Body - Gold Anodized	1
720-00004	Toddler Body - Blue Anodized	1
720-00005	Toddler Foot - Left Gold Anodized	1

720-00006	Toddler Foot - Left Blue Anodized	1
720-00007	Toddler Foot - Right Gold Anodized	1
720-00008	Toddler Foot - Right Blue Anodized	1
720-00009	Toddler Ankle	2
720-00010	Toddler Legs	4
<b>Hardware</b>		
700-00002	3/8" 4/40 machine screw – panhead	10
700-00003	4/40 nut	14
700-00016	3/8" 4/40 machine screw – flathead	6
700-00028	1/4" 4/40 machine screw – panhead	12
700-00060	1" 4/40 aluminum standoffs female/female	4
710-00100	3/16" long 4/40 socket head cap screw – stainless steel	4
712-00001	1/2" outer diameter flat round plastic washer	4
725-00002	3" long 3/16" outer diameter stainless steel rod	2
725-00003	1/16" ball joints with 2/56 thread (nut, ball joint, cup)	4
725-00004	5.4" long stainless steel rod with 2/56 0.5" thread on each end	2
725-00005	3/32" hex L-key	1
725-00006	3/32" E/Z adjust plastic horn bracket for 4-40 screw	1
725-00007	.072" brass servo horn connector (brass fitting, rubber holder and small screw) – in a package	1
725-00008	.072" diameter "L" shaped 2" wire	2
726-00001	3/16" collars (4), set screw and wrench	1
<b>Miscellaneous</b>		
122-00001	Advanced Robotics with the Toddler Manual	1
123-00001	Toddler Printed Insert	1
800-00003	Serial cable	1
900-00007	The Plastic Box	1
700-00064	Parallax Screwdriver	1

<b>Additional Recommended Hardware</b>	
Parallax Part #	Description
27312	Toddler Toes Kit for Chapter 8
28017	Memsic 2125 Accelerometer for Chapter 9

# Appendix B: Toddler PCB Schematic



<p><b>Toddler</b> Revision C.</p>	<p><b>Parallax, Inc.</b> 598 Menlo Drive, Suite 100 Rocklin, CA 95765</p>
<p>www.parallaxinc.com</p>	<p>www.stampsinclass.com</p>

**B**



## Appendix C: Blue Bell Designs Co-Processor

---

### INTRODUCING BLUE BELL DESIGN

Parallax customers make many accessories for our robot and some of them are for sale. This appendix is an example of a co-processor from Blue Bell Design ([www.bluebelldesign.com](http://www.bluebelldesign.com)). At press time for this text the Blue Bell Design product is not available through Parallax, but may be in the future..

Chapters 8 and 9 discussed Real Time Programming and some of the difficulties you can get into when trying to keep up with some task that requires servo refreshing at a relatively high rate. The Toddler challenge is keeping up with refreshing both the servos every 20 ms.



#### How does it help?

A Co-Processor is an additional processor added to the main processor, which, in Toddler's case, is the BASIC Stamp. Various manufacturers put different features in their Co-Processors. In Chapter #9, this text discussed the benefits of a ramping-type servo controller. That type of controller would perform the `Do_Movement` subroutine with little help from the main processor. The Blue Bell Design Co-Processor has ramping servo controllers for up to 8 servos, plus five analog to digital converter inputs (10-bit), and extra inputs and outputs. Best of all, it is even available with a PC board to easily attach to your Toddler.

#### Where do the interface pins come from?

Obviously, if the BASIC Stamp is to talk to this Co-Processor, it will require some I/O pins from the BASIC Stamp. In this case it's four. If you were counting the earlier programs, you'll see that the program in Chapter #8 had only a few spare pins left. Chapter #9 had even fewer left. Evaluating the pin usage carefully you can see the display LEDs can go. This isn't an altogether great solution but would work in a pinch. The pin-out of the PC board also requires moving some functions to different I/O ports but that is even less of an issue thanks to the flexibility of the new `PIN` type of variable in PBASIC 2.5. It turns out, however that the unused servo controller outputs from the Co-Processor can also be used as regular digital outputs. Since we only have two servos on Toddler, we can use servo channels 4->7 for the indicator LEDs and still have channels 2 and 3 as spare servo controllers. Maybe for a tilt and pan camera mount? Now we are

actually ahead on the pins as we used 4 Toddler I/O to get 6 Co-Processor I/O. The two Toddler freed up pins can again be used for a Memsic 2125 Accelerometer.

### **How do I drive the Servo Controller?**

The four interface lines to the Co-Processor are reset, timer complete, serial data to the Co-Processor and serial data from it. You'll need to do a co-processor reset in the initialization routine. Just pull the reset line low for a few milliseconds to do that. To tell a servo controller to go to a certain position, you send two bytes. The first byte tells the co-processor that you want a certain channel to move to a new place. The second byte tells where you want it to go. Remember we used a `WORD` variable for `PULSOUT` to drive the servos before. Here we only need a byte because, rather than being general case like `PULSOUT`, the servo controller byte defines how many steps of 4 us are added to a base value of 994 us. In other words, a 0 sent to the servo controller gives 0.994 ms. Send a 255 and get 2.014 ms. That is almost exactly the 1 to 2 ms range specified for most servos. If you need wider pulses for special applications, there is a "coarse" mode that spreads the range somewhat. Sending a total of four bytes then tells the controller where both the servos need to be and it will continue to refresh them every 20 ms with no further help from the toddler main processor.

```
SEROUT To_Co_Proc, Baud, [TiltServo,newValue]      'Set Servo 0 = Tilt
SEROUT To_Co_Proc, Baud, [StrideServo,newValue]    'Set Servo 1 = Stride
```

As we discussed in Chapter #9, you really don't want the servo to move to the final position at full speed. That is why we had `TiltStep` and `StrideStep` in the `Do_Movement` subroutine. To get the servo controllers to ramp to position, just send two bytes per servo channel. The first tells the controller to store the ramp value and enable ramping. The second value contains the ramp rate for that channel. Unlike `TiltStep` and `StrideStep`, the servo controller adds only a quarter of the value sent for each 20 milliseconds cycle (it keeps track of the fractions internally). This allows for up to a 20 second full range transition. The maximum ramped speed is 31 counts, which takes 0.66 seconds for the transition. Setting the ramp value back to zero disables ramping for that channel. Ramp rates stay in effect until rewritten or the co-processor is reset. They are also not dependent on the loop speed in the BS2 as they are incremented once per 20 ms not once each time through the control loop. Since we have only been using a single rate for tilt and another one for stride, we can write the ramp values at initialization time and not worry about it anymore. In the following code we used `TiltStep*4` to keep the steps sizes like the previous programs.



```
SEROUT To_Co_Proc, Baud, [TiltRamp,(TiltStep*4)] 'write rate to servo0
SEROUT To_Co_Proc, Baud, [StrideRamp,(StrideStep*4)] 'write rate to servo1
```

### **How do I know when the servo is finished moving?**

One benefit of calculating in the BASIC Stamp is that you constantly know where the servo is at any given time. The disadvantage is, of course, that it takes a lot of power from the processor! Since the servo controller in the co-processor is now doing this for you, you'll have to figure out how long it will take to move the servo to the new position yourself. This is easily done by finding the difference from where you are to where you are going and dividing by the step rates. Now you have almost nothing to do but wait until the servo controller is finished with the move. If you use a **PAUSE** instruction, the processor stops while it is waiting. That isn't much help! You could also read the current servo position out of the servo controller but there is a better way. Here is where a unique feature of the Blue Bell Design Co-Processor comes into play. There are 8 timer channels that keep time in increments of 20 ms. You can set a timer value and periodically check to see if any are finished by looking at the single Timer Expired pin. That pin goes high to signify a timer just completed. It is reset by reading a timer status byte that tells which timer(s) finished. Here we'll use timer 0.

```
SEROUT To_Co_Proc, Baud, [(128),DelayTime] ' write Timer 0 with delay value
```

To test the timers and reset the flag:

```
IF Timer_Exp = 1 THEN ' check if any timers done
                        ' read alarm byte to get and
clear                  '
SEROUT To_Co_Proc, Baud, [119] ' send the 1 byte command
SERIN Frm_Co_Proc, Baud, [SerDIn] ' data comes back into SerDIn

IF (SerDIn & $01) THEN doneMoveFlag = 1 ' servo is finished if Timer 0.
                                        ' If any other timers are also in
use,                                    '
                                        ' add your code to check them
here.                                    '
ENDIF ' IF Timer_Exp = 1
```

Now let's see the whole code.



```

' -----[ Title ]-----
' Toddler CoP Program A.1 FSM Walker.bs2
' This program has real time servo programming and sensor integration.
' Toddler now wanders in a straight line until it sees or hits something.
' by H. W. Lewis (www.bluebelldesign.com)
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ I/O Definitions ]-----

To_Co_Proc      PIN    2      ' pin for data to Co-Processor
Frm_Co_Proc     PIN    3      ' pin for data from Co-Processor
Timer_Exp       PIN    0      ' Timer Expired I/O pin from Co-Processor
Rst_Co_Proc     PIN    1      ' pin for reset to Co-Processor when wired

LeftIRLED       PIN    4      ' left IRLED
RightIRLED      PIN    15     ' right IRLED
LeftIR_SensorIn PIN    11     ' left IR sensor input
RightIR_SensorIn PIN    14     ' right IR sensor input

Spkr            PIN    13     ' beeper pin

LeftBumper      PIN    7      ' bumper switch on left foot
RightBumper     PIN    10     ' bumper switch on right foot

Left_B_PU       PIN    8      ' left bumper pull up - not needed unless
' 3-wire connector used on new Toddler PCB
Right_B_PU      PIN    9      ' right bumper pull up - not needed unless .."

CycleIndicator  PIN    12     ' shows how fast main loop is going - toggles
' each time through - NOT REALLY NEEDED

'EMPTY         PIN    5      ' spare Stamp I/O pin
'EMPTY         PIN    6      ' spare Stamp I/O pin

' -----[ Constants ]-----

Baud            CON    84      ' 9600 Baud => BS2 = 84
' CoProc V1.1 pin 21 pulled up = 2400 Baud
' CoProc V1.1 pin 21 grounded = 9600 Baud

TiltServo       CON    136     ' use Servo 0 = Tilt
' (136 = write setpoint command)
StrideServo     CON    137     ' use Servo 1 = Stride

StrideServoPos  CON    25     ' read tilt servo current position in Co-Proc
TiltStep        CON    3      ' TiltServo step size - was 5 steps/20ms

```

```

RightTilt      CON    40      ' Tilt limits (other Toddler used 620 = 1240us)
              ' that would be CoP = (61*4us)+ 994 = 1238us
              ' This Toddler needed 40 or 1154us
CenterTilt     CON    127     ' BS2 was 750 = 1500us
              ' CoP = (127*4us)+ 994 = 1502us
LeftTilt       CON    191     ' BS2 was 880 = 1760us
              ' CoP = (191*4us)+ 994 = 1758us

StrideStep     CON    3         ' StrideServo step size - was 5 steps/20ms

RightStride    CON    76      ' Stride limits BS2 was 650 = 1300us
              ' CoP = (76*4us)+ 994 = 1298us
CenterStride   CON    127     ' BS2 was 750 = 1500us
              ' CoP = (127*4us)+ 994 = 1502us
LeftStride     CON    176     ' BS2 was 850 = 1700us
              ' CoP = (176*4us)+ 994 = 1698us
TiltRamp       CON    144     ' start value to write servo 0 ramp
              ' (command = 128 + 16 + ch0 = 144)

StrideRamp     CON    145     ' write servo 1 ramp=1 more than servo 0 command

              ' Co-Proc servo channels 4,5,6 and 7 are used
              ' for LED drive - gives back lots to I/O pins
              ' to Stamp

'servo 4 = left bumper, 5 = rt bumper, 6 = Left vision, 7 = Right vision
' For Bumper LEDs, low = on. For vision LEDs, High = on.
' This sounds weird but it allows easy wiring on the PCB.

LeftVisionLEDOff  CON    246     ' left vision display LED (0 = Off)
              ' uses servo channel 6 output for LED drive
LeftVisionLEDon   CON    254     ' left vision display LED - 1 = On means
              ' executing vision sequence for left side

RightVisionLEDOff CON    247     ' right vision display LED - 0 = Off
              ' uses servo channel 7 output for LED drive
RightVisionLEDon  CON    255     ' right vision display LED - 1 = On means
              ' executing vision sequence for right side

LeftBumpLEDOff   CON    252     ' left bumper display LED - 1 = Off
              ' uses servo channel 4 output for LED drive
LeftBumpLEDon    CON    244     ' left bumper display LED - 0 = On means
              ' executing bumper sequence for left side

RightBumpLEDOff  CON    253     ' right bumper display LED - 1 = Off
              ' uses servo channel 5 output for LED drive
RightBumpLEDon   CON    245     ' right bumper display LED - 0 = On means
              ' executing bumper sequence for right side

```



```

Fwd      CON      0      ' codes to pick movement table
PivL     CON      1      ' set newMove to one of these values and
PivR     CON      2      ' New_Movemt will LOOKUP the right movement table
BumpL    CON      3
BumpR    CON      4

' -----[ Variables ]-----

DelayTime  VAR      Byte      ' # of 20 cycles to wait for move to complete
SerDIn     VAR      DelayTime
           ' variable where Co-Processor data comes back - reused

doTiltFlag  VAR      Bit        ' flag: 1 = tilt servo has new value
           ' flag: 0 = stride servo has new value
doneMoveFlag  VAR      Bit        ' flag: 1 = servos have reached new values
rightIR_Flag  VAR      Bit        ' flag: 1 = see something on right side
leftIR_Flag   VAR      Bit        ' flag: 1 = see something left side
rightBumpFlag VAR      Bit        ' flag: 1 = bumper hit on right side
leftBumpFlag  VAR      Bit        ' flag: 1 = bumper hit on left side

sensors     VAR      Nib        ' the lower 2 bits of the sensors variable
           ' are used to store IR detector values.
currentMove  VAR      Nib
newMove     VAR      Nib

Mx          VAR      Word       ' index for movement tables
MxOffset    VAR      Byte       ' added to Mx for index
bMovmnt     VAR      Byte       ' table value for lookup to movement

currentTilt  VAR      Byte
currentStride VAR      Byte
newValue    VAR      Byte
newStride   VAR      Byte

' -----[ Basic Movement Codes ]-----

'
' These are actual values saved in the Basic Movement tables.

TL         CON      0      ' use lower nibble for these
TC         CON      1
TR         CON      2

SL         CON      3
SC         CON      4
SR         CON      5

xx         CON      255     ' table end code

```

```

' -----[ Basic Movement Tables ]-----
'
' These tables contain Basic Movements consisting of bytes containing
' above Basic Movement Codes to describe movement sequences.
' An xx indicates the end of a list.
' Note that PivotLeft and PivotRight aren't necessarily entered at the start
' of their tables

Forward      DATA    TR, SL, TL, SR, xx
              ' possible rt bumper hit while SR
              ' or left bumper hit while SL

PivotLeft    DATA    TR, SL, TL, SR ' copy of forward, index into here
              ' by keeping MxOffset value from Forward.

              DATA    TC, SL, TL, SR, xx
              ' this is the actual pivot move.

PivotRight   DATA    TR, SL, TL, SR ' copy of forward, index into here
              ' by keeping MxOffset value from Forward.

              DATA    TR, SL, TC, SR, xx
              ' this is the actual pivot move.

RBumper      DATA    SL, TR, SR, TC, SL, TL, SR, xx
              'back up 1 step and pivot left
              ' bumper hit assumes SR when hit happened.

LBumper      DATA    SR, TL, SL, TC, SR, TR, SL, TL, SR, xx
              'back up 1 step and pivot right
              ' bumper hit assumes SL when hit happened.

' -----[ Initialize ]-----

LOW Rst_Co_Proc          'Reset Co-Processor if wired to port
PAUSE 20
HIGH Rst_Co_Proc

SEROUT To_Co_Proc, Baud, [TiltRamp, (TiltStep*4)]
                          'write rate to servo 0

SEROUT To_Co_Proc, Baud, [StrideRamp, (StrideStep*4)]
                          'write rate to servo 1

HIGH Left_B_PU          ' power left bumper pullup
HIGH Right_B_PU         ' power right bumper pullup
GOSUB Clr_Vision        ' reset vision LEDs and flags

```



```

SEROUT To_Co_Proc, Baud, [LeftBumpLEDOff]
    ' turn off LED

SEROUT To_Co_Proc, Baud, [RightBumpLEDOff]
    ' turn off LED

ResetFeet:
    CurrentTilt = CenterTilt
    CurrentStride = CenterStride

SEROUT To_Co_Proc, Baud, [TiltServo,CurrentTilt]
    'Set Servo 0 = Tilt

SEROUT To_Co_Proc, Baud, [StrideServo,CurrentStride]
    'Set Servo 1 = Stride

'DEBUG "resetCC", CR, lf

'DEBUG "Forward = ", hex Forward, cr
'DEBUG "PivotLeft = ", hex PivotLeft, cr, "PivotRight = ", hex PivotRight,
cr
'DEBUG "RBumper = ", hex RBumper, cr, "LBumper = ", hex LBumper, cr

doneMoveFlag = 1
sensors = 0
MxOffset = 0

output Spkr                ' set all I/O lines sending freqout
output LeftIRLED           ' signals to function as outputs
output RightIRLED

PAUSE 1000                 ' wait with the feet straight at startup
freqout Spkr, 2000, 3000   ' program start/restart signal.
rightBumpFlag = 0
leftBumpFlag = 0

currentMove = 15           ' start with an invalid value to assure start
' newMove = Fwd             ' for testing single moves -
' newMove = PivL           ' you have to comment out GOSUBs to vision
' newMove = PivR           ' and bump or the value might be overwritten
' newMove = BumpL
' newMove = BumpR

' -----[ Main Code ]-----

Main_Program:
DO
    TOGGLE CycleIndicator    ' indicate speed of loop on pin (typ 12 -> 14ms)

```

```

IF Timer_Exp = 1 THEN      ' check if any timers done
                          ' read timeout alarm byte to get and clear it
SEROUT To_Co_Proc, Baud, [119] ' send the byte command
SERIN Frm_Co_Proc, Baud, [SerDIn]      ' data comes back into SerDIn

'DEBUG "      DelayTime = expired, SerDIn = ",HEX SerDIn, " i = ",
      DEC i,CR

IF (SerDIn & $01) THEN doneMoveFlag = 1
                          ' servo is finished if Timer 0.
                          ' If any other timers are also in use,
                          ' add your code to check them here.
ENDIF                    ' IF Timer_Exp = 1

GOSUB Do_Vision          ' look for obstacles with IR
GOSUB Do_Bumper          ' check for obstacles with bumpers

GOSUB New_Movevt        ' generates next move from table
GOSUB Do_Movement       ' gives servo commands out
LOOP

END

' -----[ Subroutines ]-----

Do_Vision:

FREQOUT LeftIRLED,1,38500 ' pulse left IRLED.
sensors.bit0 = LeftIR_SensorIn      ' store IR detector output in RAM.

FREQOUT RightIRLED,1,38500 ' repeat for the right IR pair.
sensors.bit1 = RightIR_SensorIn

                          ' check if currently doing an IR move
IF ((leftIR_Flag = 1) & (rightIR_Flag = 1)) THEN See_Both

IF (leftIR_Flag = 1) THEN See_Left ' is left only?

IF (rightIR_Flag = 1) THEN See_Right      ' is right only?

' By loading the IR detector output values into the lower 2 bits of the
' sensors
' variable, a number between 0 and 3 that the branch command can use is
' generated.
' Sensor outputs a low to indicate it saw something so BRANCH looks
' backwards.

BRANCH sensors, [See_Both,See_Right,See_Left,See_None]

```



```

See_Both:
  newMove = PivR
  rightIR_Flag = 1           ' flag: 1 = see something on right side
  leftIR_Flag = 1           ' flag: 1 = see something left side

  SEROUT To_Co_Proc, Baud, [LeftVisionLEDon]
                          ' turn on LED

  SEROUT To_Co_Proc, Baud, [RightVisionLEDon]
                          ' turn on LED

  RETURN

See_Right:
  newMove = PivL
  rightIR_Flag = 1           ' flag: 1 = see something on right side
  SEROUT To_Co_Proc, Baud, [RightVisionLEDon]
                          ' turn on LED

  RETURN

See_Left:
  newMove = PivR
  leftIR_Flag = 1           ' flag: 1 = see something left side
  SEROUT To_Co_Proc, Baud, [LeftVisionLEDon] ' turn on LED
  RETURN

See_None:
  newMove = Fwd
  RETURN

'-----

Do_Bumper:

  ' Bumpers on each foot. A hit causes Toddler to back and turn.
  ' Code assumes leg that hit is on the leg that is moving
  ' (while leaning on the other leg).
  ' A hit backs up the hit leg right away then steps back 1 step
  ' and does a pivot away.

  IF (leftBumpFlag = 1) THEN Bump_LeftA ' check for old bumper hit on left
  IF (rightBumpFlag = 1) THEN Bump_RightA
                          ' check for old bumper hit on right

  IF (LeftBumper = 0) THEN Bump_Left ' check for bumper hit on left
  IF (RightBumper = 0) THEN Bump_Right ' check for bumper hit on right
  RETURN

Bump_Left:

```



```

'DEBUG "bump left", cr      ' sends only once on change
SEROUT To_Co_Proc, Baud, [StrideServoPos]
                          ' read the stride position
SERIN  Frm_Co_Proc, Baud, [SerDIn]  ' data comes back into SerDIn
currentTilt = SerDIn      ' set the current position there for delay
SEROUT To_Co_Proc, Baud, [LeftBumpLEDOn]
                          ' turn on LED

Bump_LeftA:
  leftBumpFlag = 1        ' flag indicates doing a bump response move
  newMove = BumpL
  GOTO Clr_Vision        ' goes to Clr_Vision which does RETURN

Bump_Right:
'DEBUG "bump right", cr
SEROUT To_Co_Proc, Baud, [StrideServoPos]
                          ' read the stride position
SERIN  Frm_Co_Proc, Baud, [SerDIn]  ' data comes back into SerDIn
currentTilt = SerDIn      ' set the current position there for delay
SEROUT To_Co_Proc, Baud, [RightBumpLEDOn]
                          ' turn on LED

Bump_RightA:
  rightBumpFlag = 1
  newMove = BumpR
                          ' falls through to Clr_Vision which does RETURN
'-----

Clr_Vision:
                          ' reset vision LEDs and flags

  IF rightIR_Flag = 1 THEN
    rightIR_Flag = 0      ' flag: 1 = see something on right side
    SEROUT To_Co_Proc, Baud, [RightVisionLEDOff]
                          ' turn off LED

  ENDIF                  ' IF used to prevent constant serial transmissions

  IF leftIR_Flag = 1 THEN
    leftIR_Flag = 0      ' flag: 1 = see something left side
    SEROUT To_Co_Proc, Baud, [LeftVisionLEDOff]
                          ' turn off LED

  ENDIF

  RETURN

'-----

New_Movemt:
                          ' output sequences for forward motion

```



```

IF (newMove <> currentMove) THEN      ' if movement changed set up new DATA
                                        ' table.

'DEBUG cr, "start NewMovemt; newMove = ", HEX newMove
'DEBUG " ", currentMove = ", HEX currentMove, cr

doneMoveFlag = 1                      ' stop current move wherever it is
currentMove = newMove

' When customizing the program, here is where to integrate between moves.
' You can make the transitions reasonable by using the existing Mx value
' and MxOffset to compute the proper MxOffset to start into the next move.

IF ((newMove = Fwd) OR (newMove = BumpL) OR (newMove = BumpR)) THEN
  MxOffset = 0                        ' reset old MxOffset value for new move
ENDIF                                 ' note: IF statement kept the old MxOffset
                                        ' value for starting pivot left or right

LOOKUP newMove, [Forward, PivotLeft, PivotRight, LBumper, RBumper], Mx
                                        ' lookup movement table index
ENDIF

IF (doneMoveFlag = 1) THEN            ' done moving - lookup new move in table
  READ (Mx + MxOffset), bMovmnt       ' read next basic move byte
'DEBUG "Mx = ", HEX Mx, ", MxOffset = ", HEX MxOffset
  MxOffset = MxOffset + 1

IF (bMovmnt < xx) THEN                ' end code so do new move
'DEBUG " bMovmnt = ", HEX bMovmnt, " "

SELECT bMovmnt                        ' set and display movement type
CASE TL
  newValue = LeftTilt
  doTiltFlag = 1
'DEBUG "TL, ", cr
CASE TC
  newValue = CenterTilt
  doTiltFlag = 1
'DEBUG "TC, ", cr
CASE TR
  newValue = RightTilt
  doTiltFlag = 1
'DEBUG "TR, ", cr
CASE SL
  newValue = LeftStride
  doTiltFlag = 0
'DEBUG "SL, ", cr
CASE SC
  newValue = CenterStride
  doTiltFlag = 0
'DEBUG "SC, ", cr

```

```

CASE SR
  newValue = RightStride
  doTiltFlag = 0
  'DEBUG "SR, ", cr
ENDSELECT
' will fall through if invalid index
ELSE
  'DEBUG " xx = finished table", cr, cr
  MxOffset = 0 ' finished move sequence so restart
  leftBumpFlag = 0 ' clear out left bumper hit flag
  rightBumpFlag = 0 ' right too
  SEROUT To_Co_Proc, Baud, [LeftBumpLEDOff]
  ' turn off left bumper display LED
  SEROUT To_Co_Proc, Baud, [RightBumpLEDOff]
  ' right too
  GOSUB Clr_Vision ' reset vision LEDs and flags
  GOSUB Do_Vision ' since done previous movement, figure out new
  ' one
  GOSUB Do_Bumper
  GOTO New_Movemt ' restart this subroutine to get new
  ' table and offset
ENDIF
ENDIF

'Done_New_Move:
RETURN ' ignore if invalid

'-----
Do_Movement:

IF (doneMoveFlag = 1) THEN ' only do this when the move is done
  DelayTime = 0

IF (doTiltFlag = 1) THEN ' only do this when the move is tilt
  IF (currentTilt < newValue) THEN
    DelayTime = newValue - currentTilt
  ELSE
    DelayTime = currentTilt - newValue
    ' coming from left tilt = decrement
  ENDIF

  SEROUT To_Co_Proc, Baud, [TiltServo,newValue]
  'Set Servo 0 = Tilt
  DelayTime = DelayTime/TiltStep
  CurrentTilt = newValue

ELSE ' IF (doTiltFlag = 1)

```



```
IF (currentStride < newValue) THEN
  DelayTime = newValue - currentStride
ELSE
  DelayTime = currentStride - newValue
ENDIF

SEROUT To_Co_Proc, Baud, [StrideServo,newValue]
                          'Set Servo 1 = Stride
DelayTime = DelayTime/StrideStep
CurrentStride = newValue

ENDIF                                ' IF (doTiltFlag = 1)

IF DelayTime < 255 THEN DelayTime = DelayTime + 1
                          ' to make up for fractions from divide
'DEBUG "      DelayTime = ", DEC DelayTime, CR,LF
SEROUT To_Co_Proc, Baud, [(128),DelayTime]
                          'command to write Timer 0 with delay value

doneMoveFlag = 0          ' reset flag

ENDIF                                ' IF (doneMoveFlag = 1)

RETURN
```

By using the `CycleIndicator` output we can see that the `Main_Program DO...LOOP` now normally takes only 12 to 14 milliseconds. More to the point, it no longer matters anyway! The servos are kept moving along by the servo controllers. We now have almost the complete BASIC Stamp 2 processor back for whatever great new things we can come up with. If the program takes a while to get back to checking for the “Timer Expired” line to be complete, it just means the response to seeing or feeling something or maybe the time between moves is delayed. The smoothness of the servo movements is not sacrificed. Programming also becomes much easier and non-critical.

### **What other features are in the Blue Bell Design Co-Processor?**

There are also six channels of 10 bit A/D. Channel 0 is permanently wired to the battery supply voltage. One of the others could be connected to the Memsic 2125 Accelerometer to allow temperature compensation from the analog temperature output. Analog outputs from IR distance sensors, temperatures, humidity, can all now be accurately measured.



There is an IR vision system and bumper sensors that can be used if BASIC Stamp I/O becomes a problem. See the data on Blue Bell Design’s website for more detailed information.

### **The easy way to more fun!**

Tuning a processor for absolute maximum performance can be a big pain, even though it’s a standard part of any embedded process. On the other hand, with a Blue Bell Design Co-Processor, you can easily try out new and fun behaviors, added sensors, or whatever, on your Toddler. Best of all, it is available on an easily attached board for convenient installation on your Toddler! Purchase it from [www.bluebelldesign.com](http://www.bluebelldesign.com).



# Index

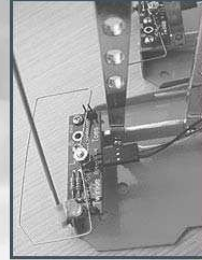
---

- A -
- Accelerometer, 189
- Audience, vi
- C -
- Co-processor, 205
- D -
- Discussion group, vi
- E -
- Educational concepts, v
- F -
- Finite State Machine (FSM)  
programming, 176, 184
- Foreign translations, vi
- G -
- Guarantee, 2
- I -
- Infrared, 113
- M -
- Memory map, 75
- Movements
  - Brute force method, 36
  - DATA tables, 50
  - State transitions, 56
  - Turns, 65
- P -
- PBASIC Commands
  - DATA, 50
  - FREQOUT, 115
  - RCTIME, 92
  - Variable aliases, 148
- Photoresistors, 89
- R -
- Real time programming, 165, 190
- S -
- Servos, 29
- Subsumption architecture programming,  
186
- T -
- Teacher's Guide, vi
- Toddler Circuit Board
  - Basic connections, 6
  - Power switch, 7
- Toddler Hardware
  - Angling feet outsteps, 21
  - Center of gravity, 28

## Prevent your Toddler Robot from running into objects using **Toddler Bumper Sensors.**



The Toddler Bumper Sensors are an add-on that attach to the feet and wire directly to the Toddler's BASIC Stamp control board. Bi-color LEDs indicate switch status.

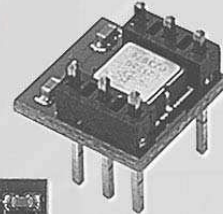


**PARALLAX**

For more information on sensors, or to order the Bumper Sensors visit [www.parallax.com](http://www.parallax.com)

# Measure Acceleration

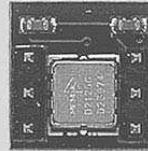
The Memsic 2125 is a low cost, dual-axis thermal accelerometer that is capable of measuring dynamic acceleration (vibration) and static acceleration (gravity) within a range of  $\pm 2g$ . For integration into existing applications, the Memsic 2125 is electrically compatible with other popular accelerometers and is easy to interface to the BASIC Stamp. Great to use in your next robotic, R/C airplane or alarm application.



Memsic 2125  
Dual-Axis Accelerometer  
#28017

**PARALLAX**

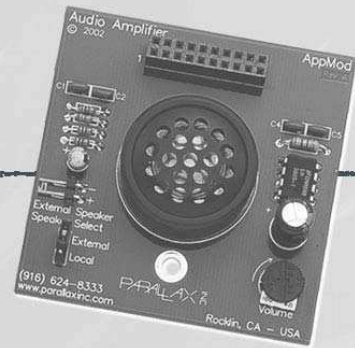
Order online at  
[www.parallax.com](http://www.parallax.com)





## ADD SOUND OUTPUT TO YOUR ROBOT!

Make the most of your BASIC Stamp's DTMFOUT and FREQOUT commands by using an Audio Amplifier Application Module. The Audio Amplifier AppMod employs an industry standard LM386 audio amplifier. Raw audio output is taken from BASIC Stamp Pin-10 and filtered prior to amplification; this converts the pulse-width modulation output of the BASIC Stamp to sinusoidal waves for the best audio quality. The amplified signal is fed to an onboard 8 ohm speaker. Order **#29143** at [www.parallax.com](http://www.parallax.com).



PARALLAX

## sensors!

Expand the capabilities of your next BASIC Stamp project with a sensor from Parallax. We stock an entire line of sensors that are compatible with the BASIC Stamp microcontroller.

*Clockwise from top:*

- TAOS TCS230 Color Sensor (#30054)
- Memsic 2125 Dual-Axis Accelerometer (#28017)
- Sensirion SHTX Humidity Sensor (#28018)
- FlexiForce Pressure Sensor (#30056)

For these and other sensors visit the Component Shop at [www.parallax.com](http://www.parallax.com)

PARALLAX

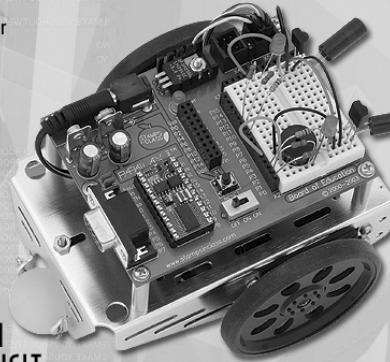


# LEARN ROBOTICS WITH THE BOE-BOT

If you enjoyed learning microcontroller programming with the BASIC Stamp, why not continue the learning curve by building a robot?

The *Robotics!* curriculum uses your BASIC Stamp 2 module and Board of Education to create a rolling robot that can follow or avoid light, detect and avoid objects with infrared, and much more.

FOR MORE INFORMATION  
OR TO ORDER ONLINE VISIT  
[WWW.PARALLAX.COM/SIC](http://WWW.PARALLAX.COM/SIC)



Robotics!  
Parts & Text  
#28154



PARALLAX

# Apply Yourself!

Looking for an all-in-one solution for your latest BASIC Stamp project? Look no further than Parallax Application Modules. These modules are designed to plug easily into any of our programming boards with a BASIC Stamp and contain a 2x10 header (Board of Education, Super Carrier Board, BS2p40 Demo Board and BASIC Stamp Activity Board). The AppMods are easy to implement and you will save valuable engineering time advancing your projects.



Compass AppMod #29113



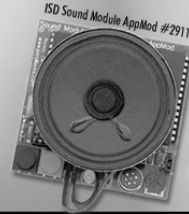
Prototype Board AppMod #29110



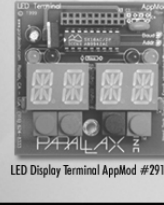
Audio Amplifier AppMod #29143



Stamp Modem AppMod #29116



ISD Sound Module AppMod #29111



LED Display Terminal AppMod #29112

Order online [www.parallax.com](http://www.parallax.com)

PARALLAX

# Go to the head of the class!

Explore microcontroller programming and interfacing with any of our popular Stamps in Class Curricula.



*What's a Microcontroller?* introduces downloading PBASIC code to a BASIC Stamp, building a circuit on a breadboard, and simple circuits with LEDs and pushbuttons. Quite easy for the beginner.



*Robotics* will teach you how to mount the Board of Education onto a robot chassis to learn mechanical control, infrared object detection, light following and avoidance.



*Elements of Digital Logic* will immerse students into a sight, sound, and hands-on logic environment. This curriculum utilizes both hardware and software simultaneously to teach logic.



*Applied Sensors* (formerly Earth Measurements) covers program structuring, sensor calibration, EEPROM data logging, conductivity of water, closed-loop feedback, and debugging in an earth science format.



The 144-page *Understanding Signals* text provides example circuits and scope configuration for viewing and capturing the common Stamps in Class signals.



The *Industrial Control* Curriculum will teach the basics of industrial process control by introducing the use of mechanical and digital switches, process control with on-off, differential gap, and PID.



*Basic Analog and Digital* will explore the principles of interfacing analog devices to digital microcontrollers. This involves the commands built into the BASIC Stamp, as well as the use of an "analog to digital converter".



*Advanced Robotics with the Toddler* will teach you how to build and program a high-quality machined two-servo bipedal walking robot controlled by an embedded BASIC Stamp 2.

**PAPALAX** [www.stampsinclass.com](http://www.stampsinclass.com)



Parts and quantities in the Toddler kits are subject to change without notice. Parts may differ from what is shown. Questions? Contact [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com).