

# Blue Bell Design Inc. Co-Processor User Manual

## Table of Contents

1.	Introduction .....	2
2.	Co-Processor to Processor Interface .....	2
2.1.	Co-Processor reset.....	3
2.2.	Serial Input (to the Co-Processor).....	3
2.3.	Serial Output (from the Co-Processor).....	4
2.4.	Timer Complete.....	4
2.5.	Baud Rate .....	4
3.	Eight Outputs .....	4
3.1.	Servo Controllers .....	5
3.2.	Fine and Coarse Mode for servos .....	5
3.3.	Servo Controllers with Ramping .....	6
3.4.	Programming for Ramping Servo Control .....	6
3.5.	Reading Servo Values .....	7
3.6.	Using the Outputs Channels as Conventional Outputs.....	7
4.	Timers.....	8
5.	A/D Converter .....	9
6.	Bumpers .....	11
7.	IRPD Vision.....	12
8.	Reading Robot Status for Bumpers and IRPD Vision.....	12
9.	Subsumption Engine .....	13
9.1.	What is a Finite State Machine? .....	13
9.2.	What is Subsumption?.....	13
9.3.	The Co-Processor Subsumption Engine .....	14
9.4.	What are those "extra" levels for?.....	23
9.5.	Why are the Stop values and Mins and Maxes programmable? .....	24
9.6.	What is the Ball Bearing Bit? .....	24
9.7.	So how do I start the Subsumption Engine in the Co-Processor? .....	25
9.8.	Show Vision .....	25
10.	Complete Programs .....	26
11.	Please, Do See the website .....	26

<b>12.</b>	<b>Legal Things.....</b>	<b>26</b>
<b>12.1.</b>	<b>Copyright 2003 Blue Bell Design Inc. ....</b>	<b>26</b>
<b>12.2.</b>	<b>Authorization .....</b>	<b>26</b>
<b>12.3.</b>	<b>Notes .....</b>	<b>26</b>
<b>12.4.</b>	<b>Life Support Policy .....</b>	<b>26</b>
<b>13.</b>	<b>Appendix A – Instruction Codes.....</b>	<b>27</b>
<b>14.</b>	<b>Appendix B – Ramp Rate Table .....</b>	<b>32</b>
<b>15.</b>	<b>Appendix C – IRPD Frequency Table .....</b>	<b>33</b>

## 1. Introduction

Blue Bell Design’s Co-Processor has some advanced standard features and some unique new features as well. The design features an easy to use programming interface. That interface and the functions it provides are the main topics of this document. Our examples will be based on using a Parallax, Inc. BASIC Stamp<sup>®</sup> BS2 as the main controller. This is not to say another controller can’t be used. Of course it can! Any processor that has a serial port can be used including a PC, a palm type organizer, almost anything!

Programming examples are shown in this manual but there are many more examples on our website [www.bluebelldesign.com](http://www.bluebelldesign.com). Check there too. It will undoubtedly give you an even better insight to the Co-Processor.

## 2. Co-Processor to Processor Interface

The Co-Processor uses from one up to four I/O pins on the main processor. They are hardware reset (pin1), serial input (pin 18), serial output (pin 17), and timer complete (pin 11).

To make the programs easier to read and modify, you can define the BASIC Stamp baudmode value as a constant. For example -

```
Baud      CON    84                ' 9600 Baud => BS2 = 84, BS2p = 240
```

All the SERIN and SEROUT commands now can reference Baud instead of putting the number in multiple places throughout the program. If you want to change the value for some reason, you only have to change it in one place. We will reference Baud in all our example programs in this manual. This is a very helpful concept and you will see we use defined constants in many of our examples. Here are the pin numbers.

To_Co_Proc	PIN	9	' data to BBDI Co-Processor(Sin)
Frm_Co_Proc	PIN	10	' data from BBDI Co-Processor(Sout)
Timer_Exp	PIN	7	' Timer Expired from Co-Processor
Rst_Co_Proc	PIN	8	' reset to Co-Processor

## 2.1. Co-Processor reset

The reset input (pin1) is kept high for normal operation. With optional Brownout Protection, the Co-Processor can reset itself on power up and in the event of system noise. Since most hobby robotic power supplies are rather noisy, the Brownout protection would constantly reset the Co-Processor. With those supplies, avoid any problems by using the No-Brownout Co-Processor and a wired connection to the Co-Processor Reset input. That way the Co-Processor can run over a wide range of input supply voltages even with substantial noise. The No-Brownout part is provided with all our kits. If you have an easily available system reset line, it can drive all the Co-Processor(s) as well as the main controller. If a system reset is not easily available, you can use a controller output to reset the Co-Processor(s). Our kit example programs use a controller pin for reset.

```
SEROUT To_Co_Proc, BAUD, [116]      ' sets line to output and leaves it high
LOW Rst_Co_Proc                      ' reset Co-Processor
PAUSE 20
HIGH Rst_Co_Proc
```

You might find that, after putting extra filtering on the power supply of your microcontroller board, the existing system reset is no longer reliable. Pressing the reset button whenever you turn power on can take care of correctly resetting. If you do add a better reset circuit for your controller, it can drive the Co-Processor reset as well and save the I/O pin from the main controller.

## 2.2. Serial Input (to the Co-Processor)

Commands are sent to the Co-Processor (pin 18) as a single serial byte in 8 bit, no parity, true format with 1 stop bit (8N1). At 9600 Baud, the Baudmode for a BS2 and BS2E is 84. For the BS2SX and BS2p it is 240. Some commands also require data, which is a second byte. Some other commands cause the Co-Processor to return a byte of data on the serial output line. Appendix A has a list of all the commands the Co-Processor accepts and how many bytes are transferred in each direction.

Command decoding is set up for easier understanding and programming. While some exceptions occur, the most significant bit tells if the command is a read or write type. The lower 7 bits can be considered a register address. Multiple channel registers, like servos, timers, A/D are grouped such that the lowest 3 bits select the specific channel. On a write type command, you usually send a second byte with the data for that command. A read type command gets back a single byte. The reset instruction is an exception since it is a read type command (decimal 116), is sent twice, and has no data returned.

Delays between sending the two bytes for a write type command are usually not needed as the Co-Processor can keep up.

The Co-Processor requires TTL/CMOS type levels or roughly 0 and +5 Volts. If you are using a PC or a another device where the serial port uses true RS-232 levels, level shifters will take normal RS-232 levels and signaling and convert them to inverted TTL type

signals for the Co-Processor. The signal polarities would then have to be inverted. Level shifters and inverters are not provided in our kits. On RS-232 systems there is no provision for reset or sensing the Timer Complete line. The timers can still be used but read the Timeout Alarm byte to test them.

### 2.3. Serial Output (from the Co-Processor)

If you don't want to use more than one timer, or read the A/D, or read servo positions, or read the IRPD or bumpers, you don't need to connect the serial output pin. You can still have eight ramping servo controllers using only 1 pin, Serial In!

Of course in most cases you probably want all the extra features you can get by using the Serial Output (pin 17) as well.

On read type commands where data is expected back, no commands should be placed between the serial output and serial input commands. It might also be a good idea to refrain from fancy formatting on the receive data to keep the turn around time quick enough to catch the data coming back from the Co-Processor.

### 2.4. Timer Complete

If you don't use the timers, you don't need to connect the Timer Complete (pin 11).

All active timers decrement every 20 milliseconds. When any active timer reaches zero, the Timer Complete line goes active (high) and the corresponding Timeout Alarm bit is set to 1. Reading the Timeout Alarm (119 or hex 77) resets the Timer Complete line to low and resets all the Timeout Alarm bits to 0.

### 2.5. Baud Rate

Pin 21 of the Co-Processor selects between 2400 and 9600 baud. Grounding the pin selects 9600 baud. Pulled up to +5 is 2400 baud.

## 3. Eight Outputs

The first things most users will want to consider are the eight outputs. They have the ability to be configured as standard outputs, standard servo controllers or "ramping" servo controllers. All eight outputs are independent as to their mode. After a normal reset, all eight outputs are configured as outputs with the output set low. Some outputs have alternate functions as well.

Outputs and Servo channels

Pin Num	Chan	Function	Alternate function
12	0	Output or Servo	Subsumption left wheel drive
13	1	Output or Servo	Subsumption right wheel drive
14	2	Output or Servo	None
15	3	Output or Servo	None
28	4	Output or Servo	None
27	5	Output or Servo	None
25	6	Output or Servo	Left Side IRPD LED (IRPD Vision on page 12)
16	7	Output or Servo	Right side IRPD LED

### 3.1. Servo Controllers

To drive a servo you send a 1 to 2 millisecond pulse every 20 milliseconds. You should keep sending the pulses to every servo whether it is to be moving or not.

It is not difficult to drive a servo with a Stamp, just issue a PULSEOUT instruction. But, like during a PAUSE, while the pulse is going out, the Stamp can't do anything else. This means each servo takes 5 - 10% of a Stamp's capability. Also, getting the 20-millisecond refresh time correct is not always so easy. You have to know when you are done executing for 20 milliseconds so you can go back to re-send all the servo pulses. This can get complicated. The good news is that the Co-Processor will do this for you so the Stamp can just keep running.

Controlling servos using the Co-Processor is easy. Send a one byte instruction to the Co-Processor to address the setpoint of the servo you want followed by a second byte to give it the new desired servo position. Send 2 bytes - that's it! The addressed servo channel will go to servo mode (if it wasn't already in servo mode), and then go directly to the new position. The Co-Processor will automatically continue to send the correct pulses every 20 milliseconds until you send a new position setpoint to change it.

This one instruction will make servo 0 go to ServVal\_0 (a BYTE variable) and stay there.

```
SEROUT To_Co_Proc, Baud, [136,ServVal_0] ' Set Servo 0 = Left side
```

Another way can be to define 136 as the start address of writing the servo setpoints and use an offset named chan. For example –

Setpoint	CON	136	' start address of write Servo Setpoints
chan	VAR	Nib	' selects servo channel to be used

So the commands to set the speed of both drive wheels of a runabout robot now would be

```
chan = 0
SEROUT To_Co_Proc, Baud, [Setpoint + chan, ServVal_0]
                                ' Set Servo 0 = Left side
chan = 1
SEROUT To_Co_Proc, Baud, [Setpoint + chan,ServVal_1]
                                ' Set Servo 1 = Right side
```

Sending a value of 127 (1.5 ms) will usually center an unmodified servo or stop a modified one.

### 3.2. Fine and Coarse Mode for servos

There are two modes of servo operation. Fine mode, the default, uses 4uSec increments from 0.994 ms (0) to 2.018 ms (255). Coarse mode uses 8 uSec increments on top of a 480 uSec base value. Coarse pulse values are then from 480uSec to 2520 uSec as data goes from 0 to 255. The mode bit is bit position 5 (value = 32) in each Servo Ramp byte.

If the bit is clear, fine mode is enabled, if it is set, coarse mode is enabled. Fine and coarse modes are separate for each channel.

Coarse mode is for overdriving the servos to get a longer than standard movement. You have to be careful because you can destroy the servos by using too large or too small a value. The values in Servo Min and Servo Max are used in coarse mode to protect the servos. They function as limit values to block out of range values from your program. One Servo Min value and one Servo Max value are used for all servo channels using coarse mode.

### 3.3. Servo Controllers with Ramping

The servos can also ramp from position to position (on unmodified servos) or from speed to speed (on modified servos like the drive wheels). Ramping is a slowed transition from one position or speed to another. It prevents the shock of abrupt position or speed changes that can cause wear and tear on your servos. It also eliminates a lot of calculation the main controller would have to do while the changes are taking place. If you needed to ramp to speed or position on a servo controller without the ramping feature, you would have to use the main processor to calculate all the intermediate positions. Only after reaching a steady speed or position could a non-ramping servo controller take over.

If you want the servo to use ramping to more slowly move to a new position (on unmodified servos) or turn slower (on modified servos), just send two more bytes. The first one addresses the ramp rate register for the channel you want to set and one more to set the ramp-value (0-31). Each pass through the 20 millisecond loop will increment or decrement the servo value by  $\text{ramp\_value}/4$  until the desired position is reached. The ramp range goes from 0.66 seconds to over 20 seconds for full swing. Appendix B has the ramp values versus change per 20 ms loop and full scale ramp times.

You can be disable ramping by writing a 0 to the ramp value for the channel (or a 32 if using Coarse Mode). Default on each channel is no ramping.

### 3.4. Programming for Ramping Servo Control

Once written, the ramp values for each channel will stay in effect until you write a new value. If you will always keep the same value for the ramping of a given channel, you can write any ramp values as part of the initialization routine. On the other hand, if you want to change it constantly that is fine too. This can be helpful in walker robots where you want to break a leg movement into steps. You can start moving slowly, move through the main part at high speed and then slow down again to finish the move.

The control byte to be written for each channel should have the most significant 2 bits set to zero. The next bit is the Coarse/Fine bit where 0 = fine. The least significant five bits are the ramping value from Appendix B.

```
Ramp          CON    144          ' starting address of write Servo Ramps
chan = 0
SEROUT To_Co_Proc, Baud, [Ramp + chan, ServVal_0] ' Set Servo 0 Ramping
```

To turn off ramping on a channel with coarse mode pulses

```
SEROUT To_Co_Proc, Baud, [Ramp + chan, 32]
                                     ' Set Servo = chan Ramping off,
                                     ' Stay in (or go into) Coarse mode
```

Ramping and Coarse mode settings are independent for each channel.

If you read back the ramp value from a channel you might find that the two most significant bits are not still zeros. Those bits are used internally to keep track of the fractional arithmetic used in ramping. They are counted through while the servo is changing. When the servo gets to its final position, the two top bits are again zeros.

### 3.5. Reading Servo Values

Since the servo controller is calculating intermediate positions while ramping, it could be necessary to be able to read where the servo is at a given time. This is handy if you are ramping a robot walker's leg and hit the bumper on something. Another case is a pan or tilt camera mount. The robot sees what it is looking for but where is the camera pointing?

Reading the Servo Position with the command byte = 24=>32 (Hex 18=>1F) gives where the servo is being commanded to go at that instant.

Setpoint	CON	136	' start address of write Servo Setpoints
ServoPos	CON	24	' start address of read Servo position
chan	VAR	Nib	' selects servo channel to be used
SerDIn	VAR	Byte	' holds data back from Co-Processor
chan = 0			
SEROUT To_Co_Proc, Baud, [ServoPos + chan]			' send command byte
SERIN Frm_Co_Proc, Baud, [SerDIn]			' data comes back into SerDIn
			' with Servo 0 position

Now, if you want to stop the leg or camera right where it is, send the current position to back to the setpoint. That sets the desired value to where the servo is now. The leg/camera will stop in place and stay there. It only takes two more bytes and they can be sent from one instruction –

```
SEROUT To_Co_Proc, Baud, [Setpoint + chan, SerDIn]
```

### 3.6. Using the Outputs Channels as Conventional Outputs.

If you prefer, you can use some or all of the outputs as regular outputs. The instructions 240-247 (hex F0-F7) will shut off servo mode on the channel addressed and force the output to zero (Low). Likewise 248-255 (hex F8-FF) will shut off servo mode and force the output to one (High). Since the data (zero or one) is contained in the instruction itself, no additional bytes need to be sent.

A constant zero to a servo may cause it to power down. A constant one may cause servo damage.

To restart from output mode back to servo mode for a given channel, just write a servo setpoint value. Any previously set ramping for that channel is still in effect.

#### **4. Timers**

One of the most unique features of the Blue Bell Co-Processor is the addition of timers. Many times in programming you would like to delay for a while. Perhaps you would like to check sensors at a certain rate, keep track of time of day, or flash an LED at a constant rate. In a BASIC Stamp processor it is easy to delay. Just use a PAUSE instruction. The only problem is the processor stops during the PAUSE. That means no sensor readings, no checking alarms, no nothing! Using our timers the Co-processor can keep track of eight independent delays for you. The program writes a delay value equal to the number of 20 ms time steps you want to wait. When any active timer finishes, the Timer\_Complete line goes active (high) and the corresponding Timeout Alarm bit is set to 1. Your program can execute a tight loop looking for Timer\_Complete to go high WHILE sensing or doing what ever you want to keep doing. When Timer\_Complete goes high your program can read the Timeout Alarm byte (119 = hex 77) to see which timers finished since you last checked. Reading the Timeout Alarm byte also resets it and resets Timer\_Complete back low.

Write a delay time to a timer with a (128 + timer number) instruction followed by the data byte. The data byte is the desired delay of the timer in 20 millisecond increments. Legal values are from 20 mSec to 5100 mSec. This command enables the timer if not previously running or resets it to the new value if it was. Timers are down counters from the value set to zero. All active timers decrement every 20 milliseconds.

Setting a timer to 0 delay will have no effect if it is idle, but if running, will cause it to immediately go idle. If idled in this manner, it will not set Timer\_Complete or the Timeout Alarm bit. If it has already timed out, the set Timeout Alarm bit will not be cleared.

Timers 0-7 stop timing and go idle when they time out or if a value of zero is sent for the delay time. Unlike the others, Timer 0 can also be a self-repeating timer. If started by writing to the Timer 0 Re-trigger Value (216 or hex D8), whenever the counter reaches 0 it sets Timer\_Complete, sets the corresponding Timeout Alarm bit and then resets itself to the original Re-trigger value. This causes it to trigger repeatedly until stopped by writing a delay value of 0 with a D8 command. This counter allows repetitive timing cycles without an error from re-synchronizing the loading of the timer to the down counting. This is nice for flashing LEDs but it is absolutely required for keeping track of time of day.

Read Timer Value (command is just the timer number) causes the Co-processor to send back a byte giving the current count of the particular timer. The byte reflects where the timer is when the command is answered.

Note there is roughly a 20 mSec unknown synchronization time from setting the timer until the counter starts down counting.

Start timer 0 in normal (one-shot) mode

```
SEROUT To_Co_Proc, BAUD, [128,10]      ' write Timer 0 Value to  
                                         ' alarm once after 200 ms
```

Start timer 0 in retrigger mode

```
SEROUT To_Co_Proc, BAUD, [216,25]     ' write Timer 0 Retrigger Value  
                                         ' alarm every 500 ms
```

You can read the Timer\_Complete and process the timers in your main loop.

```
check_timers:                          ' for Stamp to flash LED  
  
    IF Timer_Exp = 1 THEN                ' check for timer complete -  
                                         ' from here to done_timers only  
                                         ' takes about 4 ms and happens  
                                         ' once per 500 ms  
                                         ' only 1% load on a BS2  
  
    ' 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  
  
    IF (SerDIn.BIT0) THEN TOGGLE FlasherLED 'toggle LED on Port  
  
    ' add your code here if any other timers are in use too.  
  
    ENDIF                                ' timer work is done  
done_timers:
```

See [http://www.bluebelldesign.com/code\\_examples/Toddler\\_CoP\\_A\\_6\\_bs2.txt](http://www.bluebelldesign.com/code_examples/Toddler_CoP_A_6_bs2.txt) for some timer code that controls sequencing the states of a state machine.

## 5. A/D Converter

The quickest and most accurate way to read an analog voltage is with a true Analog to Digital converter, or A/D. Our Co-Processor has 5 channels of 10 bit A/D. This means the analog input voltage is broken up into 1024 discrete steps for measuring. That is 0.1% resolution. The range goes from 0 volts minimum to a maximum of the Co-Processor power supply voltage which is nominally 5.0 volts.

Our Co-Processor includes 5 channels of 10-bit Analog to Digital converters. Four are unused and are available for your sensors etc. Channel 0 is usually connected to the

power supply/2 to measure the state of the batteries. See the note below if your project has a four cell supply.

Often you only need an 8 bit value. If you want only the most significant 8 bits, you do a read to the desired channel for the first 8 bits. Write a byte and read a byte. That's it!

Here's the code–

```
A2D_val          VAR    BYTE          'returned 8 bit A/D value

read_A2D_8b:          'Read only the Most Significant 8 Bits of the A/D

SEROUT To_Co_Proc, Baud, [122]          'Read first byte of A/D channel 2
                                          ' command = 120 + channel number
SERIN Frm_Co_Proc, Baud, [A2D_val]      'Get the Most Significant 8 bits.
                                          ' into A2D_val. 0 = 0 volts, FF = 5 volts *(255/256)
```

But we actually have a 10 bit A/D

A byte only holds 8 bits so the value has to be broken into 2 bytes. You just saw how to get the most significant 8 bits. Ideally, the second byte will contain the least significant 2 bits such that they easily go into a WORD variable. The cleanest way to do that is to put those bits in the most significant bit positions of the least significant byte. When the A/D channel is read you first get the most significant byte as shown above. To get the second byte containing the last 2 LS bits, you just send the read command = 85 (hex 55). That will get the second byte of whatever A/D channel you read last. In this next code, let's try using the channel and serial command as variables. They could, of course, continue to be constants like the 8 bit code above. The only real differences needed for 10 bits are the second read command and using a word to hold the result.

```
chan             VAR    NIB          ' variable used for channel number
SerCmdOut        VAR    BYTE        ' Serial Command to the Co-Processor
adc_value        VAR    WORD        ' 10 bit A/D value left justified

read_A2D_10b:          ' Read all 10 Bits of the A/D from the
                        ' channel number in chan and load the
                        ' data into a word named adc_value.

    chan = 3          ' set channel number (0=>4 are valid)
    SerCmdOut = 120 + chan ' read first byte of A/D command

    SEROUT To_Co_Proc, Baud, [SerCmdOut] ' Read first byte of A/D chan
    SERIN Frm_Co_Proc, Baud, [adc_value.HIGHBYTE]
                        ' Get the Most Significant 8 bits.
    SEROUT To_Co_Proc, Baud, [85]        ' Now read 2nd byte of A/D
                                          ' command = 85 (decimal) or $55 (hex)
                                          ' independent of which A/D channel
```

```
SERIN Frm_Co_Proc, Baud, [adc_value.LOWBYTE]
```

```
' The data coming back is the least  
' significant 2 bits but left justified -  
' That is, they are in most significant  
' 2 bit positions.  
' The bottom 6 bits are 0.
```

After executing the code above, `adc_value = 0` for 0 volts input, or 1111 1111 1100 0000 for Processor power supply voltage\*(1023/1024) = 5\*(1023/1024) = 4.995 volts. Voltages in between will give values in between.

With a step size of only about 5 millivolts (0.005 volts) you can make a super datalogger. See our website for a program using our Libby controller board with an attached PC as the display. [http://www.bluebelldesign.com/code\\_examples/Datalog\\_BSP.txt](http://www.bluebelldesign.com/code_examples/Datalog_BSP.txt)

If A/D inputs are left unconnected (floating) they will read erratic numbers. Unused inputs can be connected to +5, ground, or, just don't read those channels.

The A/D converter reads the input voltage with respect to its power supply voltage. If your power supply is not at 5 volts or is not as stable and noise-free as the supply on our controller boards, you might get peculiar readings because of variations in the power supply voltage.

Note: Our Co-Processor add-on kit has channel 0 connected to the Logic Power input. If you don't use the on-board regulator (U2) the measurement will always show one-half Logic Power. The theory behind the battery checker is that the batteries are good when the voltage regulator is in regulation. Unfortunately four cell supplies, like on most commercial hobby robots, are not regulating during most of the battery life. This makes that feature somewhat less useful. Use an on-board regulator and provide the logic power from more than four cells if you want full functionality from the battery monitor, and much better battery life! An easy way to do it is to connect the bottom four cells to the Servo Power input of the Kit PCB and add two more cells in series to the logic supply input. Feed the controller from the Logic Supply terminal. It means you will need a double pole power switch (one pole for logic power and another for servo power) but the results will be worth it with increased battery life and project performance. For more on four cell power supplies, see our website at - [http://www.bluebelldesign.com/code\\_examples/Power1.htm](http://www.bluebelldesign.com/code_examples/Power1.htm)

## 6. Bumpers

Robots need to sense their surroundings. Bumpers are one good way of doing this. We have the 2 front bumper inputs connected to the Co-Processor. This saves 2 I/O pins on your main controller. The bumpers can also be used when the Co-Processor runs the subsumption engine. For example code to read the bumpers into a Stamp, see the "Reading Robot Status for Bumpers and IRPD Vision" section 8.

## 7. IRPD Vision

Actually this is not really vision in the sense of seeing details. Instead it is an InfraRed Proximity Detector (IRPD). Two IR LEDs take turns flashing at high speed and any reflections from the left/front or right/front are detected with a remote control type sensor. Putting out these high-speed (38KHz) pulses while testing other signals is not something many controllers do well. A separate Co-Processor is better for this service. Using our Co-Processor, the controller can get the results of the IRPD sensing by simply reading a status register.

The Write IRPD Frequency command (214 or hex D6) will change the operating frequency of the IRPD LEDs. As the frequency changes, the sensitivity of the detector changes. In our Co-Processor, the frequency is programmable over a wide range. This can be used to adjust sensitivity to distance even while the robot is moving. Because of the wide range of 15.1 KHz to over 62 KHz (in 127 steps), you could use the Co-Processor with other sensors other than the 38KHz one we use. Appendix C gives the frequency register values settings versus the actual IRLED frequencies. Notice that a frequency setting of zero will stop the IRPD Vision system from pulsing out or reading anything. A Co-Processor reset causes the frequency to be set at 0. Initialization for Robot Mode (command = 117 or hex 75) will start the IRPD at a frequency of 40.3KHz. This gives roughly 50% sensitivity from a PNA4602 sensor. After initialization, you can change the frequency to get a different sensitivity or set it for a different sensor. For example code to use the IRPD with a Stamp, see the “Reading Robot Status for Bumpers and IRPD Vision” section below. Another example is given at [http://www.bluebelldesign.com/code\\_examples/botstatus\\_bsp.txt](http://www.bluebelldesign.com/code_examples/botstatus_bsp.txt)

An extra feature of output channels 6 and 7 is an ability to act as LED indicator drivers to show if anything is currently being “seen” by the IRPD sensor. See the heading named Show Vision on page 25 for information on how to control this feature.

## 8. Reading Robot Status for Bumpers and IRPD Vision

Sending a Read Robot Status Command byte (118 or Hex 76) will cause the Co-Processor to return a byte that contains

Bit 7 = NOT USED	Unused = MS Bit
Bit 6 = see_on_right	1 = vision sees something on right
Bit 5 = see_on_left	1 = vision sees something on left
Bit 4 = first_bump	1 = bumper just hit - stop NOW (first_bump is valid only if robot mode is running)
Bit 3 = Rightbump	1 = right bumper hit
Bit 2 = Leftbump	1 = left bumper hit
Bit 1 = bstate2	Bumper Finite State Machine – see Subsumption program
Bit 0 = bstate1	Bumper Finite State Machine = LS Bit (Bumper FSM Bits are valid only if robot mode is running)

You can name the various bits in your program by defining them as SerDIn.BITx. Here is the code to read the status byte.

SEROUT To_Co_Proc, BAUD, [118]	'Send command to read robot status
SERIN Frm_Co_Proc, BAUD, [SerDIn]	'Data comes back into SerDIn

## 9. Subsumption Engine

We included an expandable Subsumption Engine in the Co-Processor. Since all the necessary sensor inputs already go to the Co-Processor, the Co-Processor has everything it needs to completely run a wheeled robot chassis. To do that, you just initialize and enable it. This allows the full power of the main controller to be used for your application! The robot program in the Co-Processor can be tailored by sending parameters from the main controller or even disabled entirely if you want the controller to have full control over the functions.

### 9.1. What is a Finite State Machine?

A Finite State Machine, abbreviated FSM, is a method of programming. It uses "states" that you can think of like steps you might be following from assembly instructions to build something.

For more information on FSMs, see the website at  
[http://www.bluebelldesign.com/FSM\\_explain.htm](http://www.bluebelldesign.com/FSM_explain.htm)

### 9.2. What is Subsumption?

Subsumption is a programming technique that is based on reflex types of reaction to sensor inputs. Essentially, if a higher priority stimulus is present, it takes control over the lower priority one. In other words, it is more important to react to bumping into something now rather than the fact that you are seeing something you might bump into later!

One easy way to program a Subsumption engine is to have a subroutine for each behavior. Define an action variable; let's call it "drive". Each subroutine (behavior) decides what it wants to do with drive and writes into the variable if (and ONLY IF) it needs some action. The subroutines are executed from lowest priority first to highest priority last. Since each subroutine has written into the same variable, the last one that wrote (i.e. the highest priority one that needed response) will actually determine the final value of drive. A subroutine that writes over the drive variable loaded by a previous subroutine is said to have its behavior "subsume" the previous behavior. When done the behavior subroutines, the program decodes what do based on the drive value. In the example program below, the decoding subroutine is named "action". Again more information is at

[http://www.bluebelldesign.com/FSM\\_explain.htm](http://www.bluebelldesign.com/FSM_explain.htm)

By now you probably figured out that a Finite State Machine is an excellent way to program a Subsumption Engine.

### 9.3. The Co-Processor Subsumption Engine

The Subsumption Engine in the Co-Processor uses the IRPD vision and the bumpers, along with servo outputs 0 (left) and 1 (right), to control the base of a hobby servo powered wheeled robot. The best way to cover most of the details of our Subsumption Engine is to show an example program that runs on a Stamp. This code isn't to be run on your controller, rather it is to show you what is already inside the Co-Processor for you to use. The example is somewhat simplified from the actual Co-Processor code. If you study it you can learn the details needed to program the Co-Processor's internal Subsumption Engine. A great deal of the information is in the comments, so be sure to read them too.

```
'{$STAMP BS2}
'{$PBASIC 2.5}
' Subsume_BoeBot1.BS2 - This shows a Subsumption Engine in the Stamp2.
' It uses the ramping servo controllers and sensors from the Co-Processor
' Copyright Blue Bell Design Inc. 2002, 2003

' NOTE: Some of this code is derived from the code for the TRaCY robot.
' TRaCY code is Copyright Dennis Clark and Harry W. Lewis 1999 and 2000

' This program takes 32% of code space and 21 of the 26 registers.
' But, it is all in the Co-Processor for you to use!
' In fairness, 6 of the registers used here (LX_Dirs, LX_Durs) aren't needed
' because their levels are disabled. However, they are available in the
' Co-Processor and were used here to show how the expansion level feature works.
'
' Of a 40 ms loop time, the synchronization wait time is about 14 ms!
' That means the basic loop is taking about 26 ms. The Stamp isn't doing any
' ramping calculations and Levels 1, 3 and 5 were disabled.
' If you totally remove Levels 1, 3, and 5 along with the synchronization
' using the timer, the program could make an 18 ms loop time.
'
' The constants for the durations were actually tuned for a 20 ms loop like
' in the Co-Processor. If you want to actually run this code in a BS2, you should
' divide them by 2. But, with a Co-Processor, why actually run this code in the BS2?

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

To_Co_Proc      PIN    11      ' pin for data to BBDI Co-Processor
Frm_Co_Proc     PIN    10      ' pin for data from BBDI Co-Processor
Timer_Exp       PIN     9      ' Timer Expired I/O pin from BBDI Co-Processor
Rst_Co_Proc     PIN     8      ' pin for reset to BBDI Co-Processor
BAUD            CON    84      ' 9600 BAUD => BS2 = 84, BS2p = 240
                  ' CoProc V1.1 pin 21 grounded = 9600 Baud

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

```

' The bot_control byte is defined below as its separate named bits for clarity
' Usually it is sent to the Co-Processor to control the Co-Processor's
' internal Subsumption Engine. In this case we are doing the Subsumption Engine
' here in the Stamp. The values below are what we would have sent to run
' Subsumption in the Co-Processor.
'
' 5 => Ball Bearing = 0; Show Vision = 1; level 5 = 0; do_bumpers = 1;
' 5 => level 3 = 0; do_vision = 1; level 1 = 0; do_bot = 1

Ball_Bearing      CON    0          ' Ball Bearing determines the relative
' directions vs servo pulse width. This program assumes the robot
' doesn't have ball bearing servos so it is set to 0. The changes
' show up in the subroutine "action".

Show_Vision       CON    1          ' If enabled shows the vision output
' on servo channels 6 and 7
level_5           CON    0          ' Level 5 is like Level 1 except 5 is
' highest priority
do_bumpers        CON    1          ' bumpers are at level 4
level_3           CON    0          ' Level 3 is like Level 1 except
' higher priority than Level 1 and vision
do_vision         CON    1          ' enable vision from the IR Proximity Detector
level_1           CON    0          ' you write the direction and duration
' for level 1 and it gets integrated
' into the Subsumption calculations.

'The lowest level is randomly wandering around. It is always enabled.

do_bot            CON    1          ' enables the Co-Proc internal
' subsumption engine.

'Values shown below are initialization values, they are reprogrammable in Co-Processor
LEFT_STOP        CON    127        'Left Stop
RIGHT_STOP       CON    127        'Right Stop
LEFT_MIN         CON    0          'Left Min
RIGHT_MIN        CON    0          'Right Min
LEFT_MAX         CON    255        'Left Max
RIGHT_MAX        CON    255        'Right Max

ramp_0           CON    24         'servo 0 ramp rate 6 counts/20ms = default
ramp_1           CON    24         'servo 1 ramp rate 6 counts/20ms = default

```

' DIRECTION VALUES WRITTEN INTO THE DRIVE REGISTER BY THE VARIOUS BEHAVIORS

'Direction, bit3, bit2, bit1, bit0

'xxyy

'xx = 00 -> left motor stopped

'xx = 01 -> left motor backward

'xx = 10 -> left motor forward

'xx = 11 -> left motor forward

'yy = right motor

'normal list follows

fd	CON	%1010	'forward = 10
rv	CON	%0101	'reverse = 5
st	CON	%0000	'stop = 0
tr	CON	%1000	'turn right = 8
tl	CON	%0010	'turn left = 2
rr	CON	%1001	'rotate right = 9
rl	CON	%0110	'rotate left = 6
bl	CON	%0100	'backup turning left = 4

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

SerDIn	VAR	Byte	'gets serial data back from Co-Proc
ServVal_0	VAR	Byte	'to load servo 0 delays
ServVal_0_Old	VAR	Byte	'remember old servo 0 delays
ServVal_1	VAR	Byte	'to load servo 1 delays
ServVal_1_Old	VAR	Byte	'remember old servo 1 delays
Drive	VAR	Byte	'each behavior subroutine ' writes the direction here
wDir	VAR	Byte	'Wander Direction
wDur	VAR	Byte	'Wander Duration
IRPDDir	VAR	Byte	'IRPD Direction
IRPDDur	VAR	Byte	'IRPD Duration
bDir	VAR	Byte	'Bump Direction
bDur	VAR	Byte	'Bump Duration
L1_Dir	VAR	Byte	'Level 1 Direction
L1_Dur	VAR	Byte	'Level 1 Duration
L3_Dir	VAR	Byte	'Level 3 Direction
L3_Dur	VAR	Byte	'Level 3 Duration
L5_Dir	VAR	Byte	'Level 5 Direction
L5_Dur	VAR	Byte	'Level 5 Duration
seed	VAR	Word	'random number seed

ir_left	VAR	SerDIn.BIT5	'1 = IR sensed something on left side
ir_right	VAR	SerDIn.BIT6	'1 = IR sensed something on right side
bumper_Left	VAR	Bit	'1 = IR sensed something on left side
bumper_Right	VAR	Bit	'1 = IR sensed something on right side
first_bump	VAR	Bit	'tells first impact for immediate stop
bstate	VAR	Nib	'holds state of bumper FSM

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

```
ServVal_0 = LEFT_STOP + 1           ' need a value different from what
ServVal_0_Old = LEFT_STOP + 2      ' is to be sent
ServVal_1 = RIGHT_STOP + 1
ServVal_1_Old = RIGHT_STOP + 2
first_bump = 0
bumper_Left = 0
bumper_Right = 0
```

```
SEROUT To_Co_Proc, BAUD, [116]     ' sets line to output and leaves it high
LOW Rst_Co_Proc                    ' Reset Co-Processor
PAUSE 20
HIGH Rst_Co_Proc                   ' Reset turns servos off
                                   ' and turns ramping off
```

' We would normally initialize for robot operation. That would start the IRPD  
' (IR Proximity Detect). Here we don't want the Subsumption Engine to start.  
' We just want to turn on the IRPD so we can look at what is happening.  
' Notice that the IRPD LEDs can still light IF you set the bot\_control.

```
SEROUT To_Co_Proc, BAUD, [214,23] 'write IR Freq Register
                                   'default = 23; max sensitivity = 27; min = 20
                                   'Writing a nonzero value enables the IRPD
                                   'period = 15.6 + 0.4*counts (us) (23 = 24.8uS)
                                   'Change the 23 to other values and see the effect
                                   ' on sensitivity.
```

```
SEROUT To_Co_Proc, BAUD, [218,$40] ' write bot_control to the Co-Processor
' to run the IRPD display LEDs but not its internal Subsumption Engine.
' Without this command the display LEDs (ch 6 and 7) won't light.
```

```
' 4 = Ball Bearing = 0; Show Vision = 1; level 5 = 0; do_bumpers = 0;
' 0 = level 3 = 0; do_vision = 0; level 1 = 0; do_bot = 0
```

```

SEROUT To_Co_Proc, BAUD, [144,ramp_0] 'write ramp rate to servo 0
SEROUT To_Co_Proc, BAUD, [145,ramp_1] 'write ramp rate to servo 1
      ' Ramp rate of x will increment or decrement
      ' servo pulse width by x/4 per 20 ms
      ' up to a max of x = 31
      ' write ramp command = 144 + ch#

SEROUT To_Co_Proc, BAUD, [216,2]      ' write Timer 0 Retrigger Value
      ' alarm every 40 ms

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

Main:      'subsumption architecture
GOSUB Rd_Sensors      ' check bumpers and vision values

IF do_bot = 1 THEN      ' subsumption engine enabled?
      'Below are the behaviors (in subroutines)
  GOSUB wander      ' random wander about subroutine
      ' Level0 = wander is lowest priority
  GOSUB Level1      ' do user subroutine
  GOSUB IRPD      ' do the IRPD subroutine (Level2)
  GOSUB Level3      ' do user subroutine (not shown)
  GOSUB bumpck      ' do the bumper subroutine (Level4)
  GOSUB Level5      ' do user subroutine (not shown)
      ' Level5 is highest priority
  GOSUB action      ' figure out servo values from
      ' results of previous subroutines
ENDIF      ' IF do_bot = 1

DO      ' for duration counters, hang
LOOP UNTIL Timer_Exp = 1      ' until timer is done (= 40 ms)

SEROUT To_Co_Proc, Baud, [119]      ' read timeout alarm byte to get and clear it
SERIN Frm_Co_Proc, Baud, [SerDIn]      ' data comes back into SerDIn - not used
      ' only using one timer so no need to decode

GOTO Main

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

Rd_Sensors:
'structure of bot_status byte back from the CoProcessor
'bstate1      VAR      BIT      ' Bumper Finite State Machine = LSB
'bstate2      VAR      BIT
'BumperFlg_Leftbmp VAR      BIT      ' bumper hit
'BumperFlg_Rightbmp VAR      BIT
'first_bump      VAR      BIT      ' bumper just hit - to stop NOW!
      ' N/A if robot mode isn't running

```

```

'see_on_left      VAR    BIT          ' vision sees something on left
'see_on_right    VAR    BIT          ' vision sees something on right
'bit7            VAR    BIT          ' NOT USED = MSB

                                ' read the bot_status from the CoProcessor

SEROUT To_Co_Proc, BAUD, [118]    ' Send the byte command to read robot status
SERIN Frm_Co_Proc, BAUD, [SerDIn] ' Data comes back into SerDIn

                                ' since ir_right is defined as SerDIn.BIT6 and
                                ' ir_left is defined as SerDIn.BIT5,
                                ' they are set by the definitions

IF SerDIn.BIT3 THEN
  bumper_Right = 1                ' 1 = hit something on right side
ENDIF                              ' The bumper bits are "sticky"
                                   ' they are reset in bumper FSM

IF SerDIn.BIT2 THEN
  bumper_Left = 1                 ' 1 = hit something on left side
ENDIF

RETURN

' -----

wander:
' The lowest level is randomly wandering around. It is always enabled
IF wDur = 0 THEN
  RANDOM seed                      'random direction
  LOOKUP (seed & %111),[fd,tl,fd,fd,fd,tr,fd,fd],wDir
  RANDOM seed                      'random duration
  wDur = (seed & %1111111)         'mask for 128 choices of duration
  IF wDir = fd THEN
    wDur = wDur + 20              'add 400ms for a minimum duration
  ELSE
    wDur = wDur & %111111        'mask cuts down to 64 choices of duration
  ENDIF
ENDIF                              ' IF wDur = 0
wDur = wDur - 1                   'decrement wander counter
drive = wDir                       'get direction
RETURN

' -----

```

```

Level1:                                     'do user subroutine
' you write the direction and duration for level 1 and it gets integrated
' into the Subsumption calculations.

IF ((level_1 = 0) AND (L1_Dur > 0)) THEN    ' enabled and not timed out?
  L1_Dur = L1_Dur - 1                       ' decrement duration counter
  drive = L1_Dir                             ' get direction too
ENDIF
RETURN

' -----

IRPD:                                       ' do IRPD Vision subroutine
IF do_vision = 1 THEN                       ' enabled?

  IF IRPDDur > 0 THEN                       ' IRPD move in progress
    IRPDDur = IRPDDur - 1                   ' decrement current move count
    drive = IRPDDir
  ENDIF                                     ' IF IRPDDur > 0

  IF ir_left THEN                           ' seeing anything?
    IF ir_right THEN
      IRPDDir = rl                          ' see both => rotate left away
      IRPDDur = 15                          ' internal Co-Processor code rotates
      drive = IRPDDir                       ' randomly either direction
    ELSE
      IRPDDir = tr                          ' left only => turn right
      IRPDDur = 9                           ' Duration
    ENDIF                                    ' IF ir_right
    drive = IRPDDir
  ELSEIF ir_right THEN
    IRPDDir = tl                            ' right only => turn left
    IRPDDur = 9
    drive = IRPDDir
  ENDIF                                     ' IF ir_left
ENDIF                                       ' IF do_vision = 1
RETURN

' -----

Level3:                                     'do user subroutine
                                     ' Level 3 is like Level 1 except higher priority
IF ((level_3 = 0) AND (L3_Dur > 0)) THEN    ' enabled and not timed out?
  L3_Dur = L3_Dur - 1                       ' decrement duration counter
  drive = L3_Dir                             ' get direction too

```

```

ENDIF
RETURN

'-----

bumpck:          'bumper subroutine (= Level4)
                 'Bumper reaction Finite State Machine (FSM)
                 ' State 0 = idle, just checking IF bumper is hit
                 ' State 1 = currently backing up from hit
                 ' State 2 = rotating away

IF do_bumpers = 0 THEN noBump          ' bumpers not enabled

IF ((SerDIn.BIT2 = 1) OR (SerDIn.BIT3 = 1)) THEN 'Being bumped now!
  IF first_bump = 0 THEN                ' new bump
    first_bump = 1                      ' capture first bump only
    SEROUT To_Co_Proc, BAUD, [152,LEFT_STOP,153,RIGHT_STOP]
    ' write positions (not setpoints) to servos 0 & 1
    ' to do an immediate stop!
    ' It is a trick. It will over-ride ramping for stop but
    ' start ramping to any new value from there.
  ENDIF                                ' IF first_bump = 1

  bDir = rv                             ' set backup while bumped and
  bDur = 18                             ' for a while (+1) after not being bumped
    ' internal CoProcessor code duration
    ' is shorter (10)if only one bumper side
    ' is hit.

  bstate = 1                            ' start or restart the state machine
ENDIF                                  ' IF ((SerDIn.BIT2 = 1) OR ..

IF bDur > 0 THEN                       ' not done current state yet,
  bDur = bDur - 1                      ' decrement bump timer
  drive = bDir                          ' set drive direction to bump
ELSE                                    ' current state finished, set next state
  IF bstate.BIT1 THEN                  ' was state 2
    bstate = 0                        ' end state 2, reset state machine to idle
    ' internal CoProcessor code keeps spinning
    ' if both "eyes" still see something
  ENDIF                                ' IF bstate.BIT2
  IF bstate.BIT0 THEN                  ' finished backing up = end of state 1
    IF bumper_Left THEN
      bDir = rr                        ' rotate right away from left bump
    ELSE
      bDir = rl                        ' rotate left away from right bump
    ENDIF                              ' IF bumper_Left = 1

```

```

bDur = 25
first_bump = 0
bumper_Left = 0
bumper_Right = 0
bstate = 2
drive = bDir
ENDIF
ENDIF

noBump:
RETURN

' -----

Level5:
    ' Level 5 is like Level 1 except highest priority

IF ((level_5 = 0) AND (L5_Dur > 0)) THEN
    L5_Dur = L5_Dur - 1
    drive = L5_Dir
ENDIF
RETURN

' -----

action:
    ' moves servo motors
    ' uses ramping servo controller for simplicity
    ' servo 0 = Left, Servo 1 = Right
    ' Ball Bearing assumed to be 0 (for BoeBot)

ServVal_0 = LEFT_STOP
IF drive.BIT3 = 1 THEN
    ServVal_0 = LEFT_MAX
ELSEIF drive.BIT2 = 1 THEN
    ServVal_0 = LEFT_MIN
ENDIF

IF (ServVal_0 <> ServVal_0_Old) THEN
    SEROUT To_Co_Proc, BAUD, [136,ServVal_0]
    ServVal_0_Old = ServVal_0
ENDIF

```

```

' internal CoProcessor code also has
' random direction if both bumpers hit
' internal CoProcessor bDur is random
' from 0.2 to 0.5 seconds
' reset sticky bits

' next state
' set drive direction to bump
' IF bstate.BIT1
' IF bDur > 0

```

```

'do user subroutine

```

```

' enabled and not timed out?
' decrement duration counter
' get direction too

```

```

' do left servo
' move left motor forward
' MIN and MAX reverse when BB = 1
' move left motor backward
' MIN and MAX reverse when BB = 1
'IF drive.BIT2

```

```

' Only need to send servo data
' IF different from last time.
' Set Servo 0 = Left side
' remember value sent
'IF (ServVal_0 <>

```

```

ServVal_1 = RIGHT_STOP           ' do right servo
IF drive.BIT1 = 1 THEN           ' move right motor forward
  ServVal_1 = RIGHT_MIN         ' MIN and MAX reverse when BB = 1
ELSEIF drive.BIT0 = 1 THEN      ' move right motor backward
  ServVal_1 = RIGHT_MAX        ' MIN and MAX reverse when BB = 1
                                ' MIN MAX are also reversed to reverse
                                ' servo directions on different sides
ENDIF                            ' IF drive.BIT0 = 1

IF (ServVal_1 <> ServVal_1_Old) THEN 'Only need to send servo data
                                ' IF different from last time.
  SEROUT To_Co_Proc, BAUD, [137,ServVal_1] ' Set Servo 1 = Right side
  ServVal_1_Old = ServVal_1      'remember value sent
ENDIF
RETURN                            'IF (ServVal_1 <>

END

```

Would you want to run this code? Probably not. It is shown and documented here so you can get an understanding of the Co-Processor's Subsumption Engine. This is a partial set of the code that is in our Co-Processor already. It is only waiting for you to enable it. Now you know what you are enabling when you turn it on. Your Stamp is then free to do other tasks.

If you want to run your own Subsumption Engine in the Stamp, you can. That example code could be modified to get down to an 18- 20 ms loop time.

With a servo controller, why still care about the loop time? It affects how long the behavior durations last. On the other hand, since you have the Co-Processor, the Stamp code could use some of the 8 timers rather than counting down durations in the Stamp. That has the added, and very significant, advantage of no longer requiring the Stamp code to be locked to a 20 ms cycle. The Co-Processor keeps the duration counters consistent and accurate while you concentrate on programming any added behaviors in the Stamp. The subsumption example didn't use the timers because the internal timing of the Co-Processor is locked to 20 ms and this code shows the internals better.

#### 9.4. What are those "extra" levels for?

The level names correspond to those in our Co-Processor's Subsumption Engine. Real power also comes in the unassigned levels. Level 1 lets you subsume the random behavior. If you want to always try to go straight unless higher behaviors (IRPD vision, level 3, bumper, level 5) overrule it, write the Level 1 Direction (199) = forward (5) and load Level 1 Duration (200) with the maximum value (255).

```
SEROUT To_Co_Proc, BAUD, [199, 5, 200,255]
```

You can use one of the Co-Processor timers to warn you to update the duration value before it gets counted down to zero. This gives the robot a "persistent attitude" of trying to go forward. It still responds to IRPD and bumpers but, when nothing is in its way, it always goes forward. The direction values are all defined as constants in the subsumption program above. Durations are in steps of 20 ms.

Notice Level 3 is above IRPD but below the bumpers. More vision type devices like SONAR and distance measuring IR devices could go here.

Level 5 is the highest priority. It would work well for more bumpers - like on the sides and back of the robot. And how about some down-looking sensors to make sure you don't unexpectedly visit the bottom of the stairs or run out of a Sumo ring!

These levels give you total control over the final Subsumption Engine behavior without having to duplicate all the code already in the controller. If you have 2 behaviors to put into one level, simply run the behaviors as 2 controller subroutines and write the output directions from each to the same variable. Send that variable's final value to the level you want to control from both behaviors. Last one to run is the higher priority. Hey, even more Subsumption!

### **9.5. Why are the Stop values and Mins and Maxes programmable?**

Sometimes on a modified servo, the stop point isn't actually at 1.5 ms pulse width. Once you find what the stop value should actually be, you can just change the Servo Stop value (after the initialization) to the value you need. No more pulling apart the servos to fix a minor problem! Stop values are commands 207 and 208 (hex CF and D0).

Different modified servos can have different "proportional bands". This is the range of pulsewidth where the servo changes speeds in response to a changing pulsewidth. At both extremes of pulsewidth, the speed is already at maximum in either direction. When using ramping, if the point you start to move from is too far beyond the "proportional band", it takes a while to ramp through pulsewidths where the speed does not change. This acts like a delay in the response. By changing the values of Left Min and Left Max or Right Min and Right Max (209 => 212 or Hex D1 => D4), you can have the Subsumption Engine use values closer to the ends of the "proportional band" so there is less delay. For more discussion on fundamentals of servos see our website at [http://www.bluebelldesign.com/servo\\_explain.htm](http://www.bluebelldesign.com/servo_explain.htm).

### **9.6. What is the Ball Bearing Bit?**

We called this the Ball Bearing bit because the servos we had with ball bearings ran opposite from those without them. It is more likely a servo brand difference rather than a ball bearing issue. In any case, some modified servos turn one way for a 1 ms pulsewidth and some turn the other way. If your Subsumption Engine runs the wheels backwards, change the value of the Ball Bearing bit loaded in the Robot Control byte.

## 9.7. So how do I start the Subsumption Engine in the Co-Processor?

Very simple! Do an initialization for robot mode followed by a write to the Robot Control byte. It can be as simple as just sending the three bytes –

```
SEROUT To_Co_Proc, BAUD, [117, 218, $55]
```

The code block below has a lot of explanation comments included. They are not needed but they make it easier to remember the bit meanings when you are modifying your program. They also include the startup reset code

```
' -----[ Initialization Code ]-----  
  
init:  
SEROUT To_Co_Proc, BAUD, [116]           ' sets line to output and leaves it high  
LOW Rst_Co_Proc                          ' Reset Co-Processor  
PAUSE 20  
HIGH Rst_Co_Proc  
  
SEROUT To_Co_Proc, BAUD, [117]           ' initialize for robot mode  
  
' Write bot_control - this byte controls the Subsumption Architecture.  
' The priorities from lowest to highest are shown below.  
' The lowest level is randomly wandering around.  
' Just above that is level 1, you write the direction and duration for level 1  
' into the co-Processor registers and it gets integrated into the Subsumption  
' calculations.  
' Then vision from the IR Proximity Detector.  
' Level 3 is like Level 1  
' bumpers next,  
' Finally the highest priority is level 5.  
' The bits in the bot_control byte enable each of the levels when they are a 1.  
' Ball Bearing determines the relative directions vs pulse width. BoE-Bot doesn't  
' have ball bearing servos so it is set to 0. If it is incorrectly set, the robot  
' runs backward.  
' do_bot enables the subsumption engine.  
  
SEROUT To_Co_Proc, BAUD, [218]           ' command to write bot_control  
SEROUT To_Co_Proc, BAUD, [$55]           ' run the robot!  
' 5 = Ball Bearing = 0; Show Vision = 1; level 5 = 0; do_bumpers = 1;  
' 5 = level 3 = 0; do_vision = 1; level 1 = 0; do_bot = 1
```

## 9.8. Show Vision

If the “Show Vision” bit in the Robot Control byte is set, output channels 6 and 7 will act as LED drivers to show what the Co-Processor IRPD is “seeing”. The bit has to be zero to use the channels as regular outputs or servo outputs. The default at reset is zero.

## **10. Complete Programs**

You will find many complete programs on our website. Some to start with are –  
[http://www.bluebelldesign.com/code\\_examples/BoE\\_CoP\\_B\\_bs2.txt](http://www.bluebelldesign.com/code_examples/BoE_CoP_B_bs2.txt) = BoE-Bot  
[http://www.bluebelldesign.com/code\\_examples/Toddler\\_CoP\\_A\\_1\\_bs2.txt](http://www.bluebelldesign.com/code_examples/Toddler_CoP_A_1_bs2.txt) = Toddler  
[http://www.bluebelldesign.com/code\\_examples/Toddler\\_CoP\\_A\\_6\\_bs2.txt](http://www.bluebelldesign.com/code_examples/Toddler_CoP_A_6_bs2.txt) = Toddler  
[http://www.bluebelldesign.com/code\\_examples/botstatus\\_bsp.txt](http://www.bluebelldesign.com/code_examples/botstatus_bsp.txt) = Status Register  
[http://www.bluebelldesign.com/code\\_examples/Herbert\\_bsp.txt](http://www.bluebelldesign.com/code_examples/Herbert_bsp.txt) = Android Head

## **11. Please, Do See the website**

We tried to make this manual as complete as possible but there are a lot of features to cover. If you have any questions, take a look at our website. There is a lot more information there. If you still can't find what you need to know, email to [harry@bluebelldesign.com](mailto:harry@bluebelldesign.com).

## **12. Legal Things**

### **12.1. Copyright 2003 Blue Bell Design Inc.**

#### **12.2. Authorization**

Any person is hereby authorized to view, copy, print and distribute any portion of this document subject to the following conditions:

1. The document may be used for informational purposes only;
2. The document may only be used for non-commercial purposes except by Blue Bell Design Inc. and its Dealers and Distributors; and
3. Any copy of the document or portion thereof must include the above copyright notice.

#### **12.3. Notes**

Any product, process, or technology described in the document may be subject to other intellectual property rights reserved by Blue Bell Design Inc. and are not licensed here under.

This document could include technical inaccuracies or typographical errors. Changes are regularly made to the information contained in this document. Changes may or may not be included in the future editions of the document. Blue Bell Design Inc. and its distributors may make improvements and/or changes in the Products, Processes, Technology, Descriptions, and/or Programs described in the document at any time. This document is provided "As Is" without warranty of any kind, either express or implied, including, but not limited to, any implied warranty or merchantability, fitness for a particular purpose, or non-infringement.

#### **12.4. Life Support Policy**

Blue Bell Design Inc. products are not authorized for use as critical components in life support devices or systems. As used herein:

1. Life support devices or systems are those which (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform properly can be reasonably expected to result in significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

### 13. Appendix A – Instruction Codes

Dec Value	Hex Value	Read/Write		Bytes Rcvd by Co-P	Bytes Xmit from Co-P	Default Value after reset Norm/Robot
0	00	Read	Timer Value - Timer # 0	1	1	0
1	01	Read	Timer Value - Timer # 1	1	1	0
2	02	Read	Timer Value - Timer # 2	1	1	0
3	03	Read	Timer Value - Timer # 3	1	1	0
4	04	Read	Timer Value - Timer # 4	1	1	0
5	05	Read	Timer Value - Timer # 5	1	1	0
6	06	Read	Timer Value - Timer # 6	1	1	0
7	07	Read	Timer Value - Timer # 7	1	1	0
8	08	Read	Servo Setpoint - (Left) Chan # 0	1	1	127/Left Stop
9	09	Read	Servo Setpoint - (Right) Chan # 1	1	1	127/Right Stop
10	0A	Read	Servo Setpoint - Channel # 2	1	1	127 = 1.5ms
11	0B	Read	Servo Setpoint - Channel # 3	1	1	127 = 1.5ms
12	0C	Read	Servo Setpoint - Channel # 4	1	1	127 = 1.5ms
13	0D	Read	Servo Setpoint - Channel # 5	1	1	127 = 1.5ms
14	0E	Read	Servo Setpoint - Channel # 6	1	1	127 = 1.5ms
15	0F	Read	Servo Setpoint - Channel # 7	1	1	127 = 1.5ms
16	10	Read	Servo Ramp - (Left) Chan # 0	1	1	0/24
17	11	Read	Servo Ramp - (Right) Chan # 1	1	1	0/24
18	12	Read	Servo Ramp - Channel # 2	1	1	0
19	13	Read	Servo Ramp - Channel # 3	1	1	0
20	14	Read	Servo Ramp - Channel # 4	1	1	0
21	15	Read	Servo Ramp - Channel # 5	1	1	0
22	16	Read	Servo Ramp - Channel # 6	1	1	0
23	17	Read	Servo Ramp - Channel # 7	1	1	0
24	18	Read	Servo Position - (Left) Chan # 0	1	1	127 = 1.5ms
25	19	Read	Servo Position - (Right) Chan # 1	1	1	127 = 1.5ms
26	1A	Read	Servo Position - Channel # 2	1	1	127 = 1.5ms
27	1B	Read	Servo Position - Channel # 3	1	1	127 = 1.5ms
28	1C	Read	Servo Position - Channel # 4	1	1	127 = 1.5ms
29	1D	Read	Servo Position - Channel # 5	1	1	127 = 1.5ms
30	1E	Read	Servo Position - Channel # 6	1	1	127 = 1.5ms
31	1F	Read	Servo Position - Channel # 7	1	1	127 = 1.5ms
32-63	20-3F		Undefined instructions	1	0	

Appendix A

Dec Value	Hex Value	Read/Write		Bytes Rcvd by Co-P	Bytes Xmit from Co-P	Default Value after reset Norm/Robot
64	40	Read	Drive	1	1	0
65	41	Read	Wander Direction	1	1	Forward
66	42	Read	Wander Duration	1	1	20
67	43	Read	IRPD Direction	1	1	0
68	44	Read	IRPD Duration	1	1	0
69	45	Read	Bump Direction	1	1	0
70	46	Read	Bump Duration	1	1	0
71	47	Read	Level 1 Direction	1	1	0
72	48	Read	Level 1 Duration	1	1	0
73	49	Read	Level 3 Direction	1	1	0
74	4A	Read	Level 3 Duration	1	1	0
75	4B	Read	Level 5 Direction	1	1	0
76	4C	Read	Level 5 Duration	1	1	0
77	4D	Read	Servo Min	1	1	65
78	4E	Read	Servo Max	1	1	190
79	4F	Read	Left Stop	1	1	127 = 1.5ms
80	50	Read	Right Stop	1	1	127 = 1.5ms
81	51	Read	Left Min	1	1	0 = 1ms
82	52	Read	Right Min	1	1	0 = 1ms
83	53	Read	Left Max	1	1	255 = 2ms
84	54	Read	Right Max	1	1	255 = 2ms
85	55	Read	2nd Byte A/D value	1	1	0
86	56	Read	IRPD Frequency	1	1	0/23
87	57	Read	Gen Purpose Memory Loc # 87	1	1	0
88	58	Read	Timer 0 Retrigger Value	1	1	0
89	59	Read	Counter Enables	1	1	0
90	5A	Read	Robot Control	1	1	0
91-115	5B-73	Undefined instructions		1	0	
116	74	Read	Reset	1	0	
117	75	Read	Initialize for Robot Mode	1	0	
118	76	Read	Robot Status	1	1	0
119	77	Read	Timeout Alarm	1	1	0
120	78	Read	Read 1st Byte A/D Channel # 0	1	1	
121	79	Read	Read 1st Byte A/D Channel # 1	1	1	
122	7A	Read	Read 1st Byte A/D Channel # 2	1	1	
123	7B	Read	Read 1st Byte A/D Channel # 3	1	1	
124	7C	Read	Read 1st Byte A/D Channel # 4	1	1	
125	7D	Undefined instruction		1	0	
126	7E	Undefined instruction		1	0	
127	7F	Undefined instruction		1	0	

Appendix A

Dec Value	Hex Value	Read/Write		Bytes Rcvd by Co-P	Bytes Xmit from Co-P	Default Value after reset Norm/Robot
128	80	Write	Timer Value - Timer # 0	2	0	0
129	81	Write	Timer Value - Timer # 1	2	0	0
130	82	Write	Timer Value - Timer # 2	2	0	0
131	83	Write	Timer Value - Timer # 3	2	0	0
132	84	Write	Timer Value - Timer # 4	2	0	0
133	85	Write	Timer Value - Timer # 5	2	0	0
134	86	Write	Timer Value - Timer # 6	2	0	0
135	87	Write	Timer Value - Timer # 7	2	0	0
136	88	Write	Servo Setpoint - (Left) Chan # 0	2	0	127/Left Stop
137	89	Write	Servo Setpoint - (Right) Chan # 1	2	0	127/Right Stop
138	8A	Write	Servo Setpoint - Channel # 2	2	0	127 = 1.5ms
139	8B	Write	Servo Setpoint - Channel # 3	2	0	127 = 1.5ms
140	8C	Write	Servo Setpoint - Channel # 4	2	0	127 = 1.5ms
141	8D	Write	Servo Setpoint - Channel # 5	2	0	127 = 1.5ms
142	8E	Write	Servo Setpoint - Channel # 6	2	0	127 = 1.5ms
143	8F	Write	Servo Setpoint - Channel # 7	2	0	127 = 1.5ms
144	90	Write	Servo Ramp - (Left) Chan # 0	2	0	0/24
145	91	Write	Servo Ramp - (Right) Chan # 1	2	0	0/24
146	92	Write	Servo Ramp - Channel # 2	2	0	0
147	93	Write	Servo Ramp - Channel # 3	2	0	0
148	94	Write	Servo Ramp - Channel # 4	2	0	0
149	95	Write	Servo Ramp - Channel # 5	2	0	0
150	96	Write	Servo Ramp - Channel # 6	2	0	0
151	97	Write	Servo Ramp - Channel # 7	2	0	0
152	98	Write	Servo Position - (Left) Chan # 0	2	0	127/Left Stop
153	99	Write	Servo Position - (Right) Chan # 1	2	0	127/Right Stop
154	9A	Write	Servo Position - Channel # 2	2	0	127 = 1.5ms
155	9B	Write	Servo Position - Channel # 3	2	0	127 = 1.5ms
156	9C	Write	Servo Position - Channel # 4	2	0	127 = 1.5ms
157	9D	Write	Servo Position - Channel # 5	2	0	127 = 1.5ms
158	9E	Write	Servo Position - Channel # 6	2	0	127 = 1.5ms
159	9F	Write	Servo Position - Channel # 7	2	0	127 = 1.5ms
160-191	A0-BF		Undefined instructions	1	0	

Appendix A

Dec Value	Hex Value	Read/Write		Bytes Rcvd by Co-P	Bytes Xmit from Co-P	Default Value after reset Norm/Robot
192	C0	Write	Drive	2	0	0
193	C1	Write	Wander Direction	2	0	Forward
194	C2	Write	Wander Duration	2	0	20
195	C3	Write	IRPD Direction	2	0	0
196	C4	Write	IRPD Duration	2	0	0
197	C5	Write	Bump Direction	2	0	0
198	C6	Write	Bump Duration	2	0	0
199	C7	Write	Level 1 Direction	2	0	0
200	C8	Write	Level 1 Duration	2	0	0
201	C9	Write	Level 3 Direction	2	0	0
202	CA	Write	Level 3 Duration	2	0	0
203	CB	Write	Level 5 Direction	2	0	0
204	CC	Write	Level 5 Duration	2	0	0
205	CD	Write	Servo Min	2	0	65 = 1ms
206	CE	Write	Servo Max	2	0	190 (= 2ms)
207	CF	Write	Left Stop	2	0	127 (= 1.5ms)
208	D0	Write	Right Stop	2	0	127 (= 1.5ms)
209	D1	Write	Left Min	2	0	0 (= 1ms)
210	D2	Write	Right Min	2	0	0 (= 1ms)
211	D3	Write	Left Max	2	0	255 (= 2ms)
212	D4	Write	Right Max	2	0	255 (= 2ms)
213	D5	Write	2nd Byte A/D value	2	0	0
214	D6	Write	IRPD Frequency	2	0	0/23
215	D7	Write	Gen Purpose Memory Loc # 87	2	0	0
216	D8	Write	Timer 0 Retrigger Value	2	0	0
217	D9	Write	Counter Enables	2	0	0
218	DA	Write	Robot Control	2	0	0
219-239	DB-EF		Undefined instructions	1	0	

## Appendix A

Dec Value	Hex Value	Read/Write		Bytes Rcvd by Co-P	Bytes Xmit from Co-P	Default Value after reset Norm/Robot
240	F0	Write	Set Output Channel 0 = 0	1	0	0
241	F1	Write	Set Output Channel 1 = 0	1	0	0
242	F2	Write	Set Output Channel 2 = 0	1	0	0
243	F3	Write	Set Output Channel 3 = 0	1	0	0
244	F4	Write	Set Output Channel 4 = 0	1	0	0
245	F5	Write	Set Output Channel 5 = 0	1	0	0
246	F6	Write	Set Output Channel 6 = 0	1	0	0
247	F7	Write	Set Output Channel 7 = 0	1	0	0
248	F8	Write	Set Output Channel 0 = 1	1	0	0
249	F9	Write	Set Output Channel 1 = 1	1	0	0
250	FA	Write	Set Output Channel 2 = 1	1	0	0
251	FB	Write	Set Output Channel 3 = 1	1	0	0
252	FC	Write	Set Output Channel 4 = 1	1	0	0
253	FD	Write	Set Output Channel 5 = 1	1	0	0
254	FE	Write	Set Output Channel 6 = 1	1	0	0
255	FF	Write	Set Output Channel 7 = 1	1	0	0

### Notes:

Writing a 0 or 1 to an output channel stops servo mode in that channel.

Writing a Servo Setpoint turns on servo mode for that output.

Servo Ramp = 0 means no ramping.

Servo Ramp value is 4 times (counts per 20ms) e.g. Ramp = 24 gives 6 counts per 20ms.

Setting Servo Ramp bit 5 (value = 32) gives course mode.

In Servo Coarse mode, 65 = 1ms & 190 = 2ms

Servo Min and Servo Max are used in coarse mode only.

Writing a Timer Value >0 starts that timer. Writing a 0 stops it without alarming.

Timeout Alarm resets to 0 after being read. 119(hex 77)

Write Timer 0 Retrigger Value 216 (D8) >0 starts Timer #0 in Retrigger mode.

Kit has A/D Channel # 0 wired to Logic Power/2

If A/D inputs 1=>4 are left floating they will read erratic numbers.

A Read A/D instruction writes 2nd Byte A/D value from channel read.

IRPD Frequency: 0 = off, 23 = 40.3KHz – See Appendix C for a table.

Robot mode writes Wander, IRPD, & Bump Directions and Durations.

## 14. Appendix B – Ramp Rate Table

Ramping takes from 0.66 to 20.4 seconds for full range. The ramp value is in 0.25 step increments per 20 milliseconds frame. A value of zero disables ramping and the setpoint value gets immediately sent as the new position.

### Servo Speed Chart

0 =	ALL	units/frame (no ramping - As fast as servo will allow)
1 =	0.25	units/frame (20.40 sec full range)
2 =	0.5	units/frame (10.20 sec full range)
3 =	0.75	units/frame (6.80 sec full range)
4 =	1	units/frame (5.10 sec full range)
5 =	1.25	units/frame (4.08 sec full range)
6 =	1.5	units/frame (3.40 sec full range)
7 =	1.75	units/frame (2.91 sec full range)
8 =	2	units/frame (2.55 sec full range)
9 =	2.25	units/frame (2.27 sec full range)
10 =	2.5	units/frame (2.04 sec full range)
11 =	2.75	units/frame (1.85 sec full range)
12 =	3	units/frame (1.70 sec full range)
13 =	3.25	units/frame (1.57 sec full range)
14 =	3.5	units/frame (1.46 sec full range)
15 =	3.75	units/frame (1.36 sec full range)
16 =	4	units/frame (1.28 sec full range)
17 =	4.25	units/frame (1.20 sec full range)
18 =	4.5	units/frame (1.13 sec full range)
19 =	4.75	units/frame (1.07 sec full range)
20 =	5	units/frame (1.02 sec full range)
21 =	5.25	units/frame (0.97 sec full range)
22 =	5.5	units/frame (0.93 sec full range)
23 =	5.75	units/frame (0.89 sec full range)
24 =	6	units/frame (0.85 sec full range)
25 =	6.25	units/frame (0.82 sec full range)
26 =	6.5	units/frame (0.78 sec full range)
27 =	6.75	units/frame (0.76 sec full range)
28 =	7	units/frame (0.73 sec full range)
29 =	7.25	units/frame (0.70 sec full range)
30 =	7.5	units/frame (0.68 sec full range)
31 =	7.75	units/frame (0.66 sec full range)

## 15. Appendix C – IRPD Frequency Table

Count	Period	KHz	
0			IRPD is off when the register is loaded with 0.
1	16.0	62.5	
2	16.4	61.0	
3	16.8	59.5	Period = 15.6us + (Count*0.4us)
4	17.2	58.1	Frequency = 1/Period
5	17.6	56.8	
-----			Intermediate values are not shown but are functional.
16	22.0	45.5	
17	22.4	44.6	Sensor sensitivity
18	22.8	43.9	(from PNA4602 datasheet
19	23.2	43.1	straight line curve approximation)
20	23.6	42.4	0.125424
21	24.0	41.7	22% 0.266667
22	24.4	41.0	35% 0.403279
23	24.8	40.3	<b>50% 0.535484 default setting in robot mode</b>
24	25.2	39.7	60% 0.663492
25	25.6	39.1	82% 0.7875
26	26.0	38.5	95% 0.907692
27	26.4	37.9	100% 0.975758
28	26.8	37.3	0.862687
29	27.2	36.8	0.752941
30	27.6	36.2	0.646377
31	28.0	35.7	0.542857
32	28.4	35.2	0.442254
33	28.8	34.7	0.344444
34	29.2	34.2	0.249315
35	29.6	33.8	0.156757
36	30.0	33.3	0.066667
37	30.4	32.9	
38	30.8	32.5	
39	31.2	32.1	
-----			Intermediate values are not shown but are functional.
119	63.2	15.8	
120	63.6	15.7	
121	64.0	15.6	
122	64.4	15.5	
123	64.8	15.4	
124	65.2	15.3	
125	65.6	15.2	
126	66.0	15.2	
127	66.4	15.1	