

Inside the Isaac 16 Default Program

Authored by:

Branden Gunn

Plymouth North High School Robotics Program, 1996 – 1999

Stamp Applications Intern, Parallax Inc., Summer 2000

ElectroMechanical Intern, IRobot Corp. Summer 2001

ElectroMechanical Engineering Student, Wentworth Inst. Of Tech., 1999 – 2004

Edited for BattleBots IQ By Michael Bastoni

The Isaac 16 is a Robot Control System

The Isaac16 Robot Control System uses 2 computers. The main computer, called the master microprocessor, does all of the actual work. It is up to the other processor, the Basic Stamp, to tell the master processor what to do. As a programmer, you have no control over the master processor itself, but you can still tell it what to do using the Basic Stamp. The default program that comes with the Isaac16 is long, but it is a good template to use for programs you create.

Let's learn a little about the stamp...

Stamp Math and Numbering Systems

The stamp does not use numbers like humans, it doesn't use *base 10* numbers. Humans today count using base 10, meaning that we have 10 different symbols for our numbers. When we write two numbers next to each other, such as 37, we know that it really means 3 tens plus 7 ones. Stamps count using *base 2*, which means it only has two symbols to use, namely 0 and 1. When a Stamp writes numbers next to each other, it is said to be using *binary numbers*. Binary numbers and base 2 numbers are the same thing. Humans use *decimal numbers*. Humans however, aren't used to binary numbers, so we need to convert them to decimal to understand them.

Here is an example of how the number 154 (decimal) = 10011010 (binary).

Example Binary	1	0	0	1	1	0	1	0
Place Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
Example Value	128	0	0	16	8	0	2	0

When looking at a binary number, it may seem rather long. The digit, or bit, that is furthest to the left is called the Most Significant Bit, or **MSB**. The last bit is called the Least Significant Bit, or **LSB**. In our example above, the binary number we are using is %10011010. The percent sign is used to indicate the following number is in binary. *Hexadecimal*

numbers are base 16, and are indicated with a dollar sign. We won't be dealing with hexadecimal numbers here. A number without a symbol in front of it is always a decimal number. To convert the binary number to decimal, we multiply each binary digit by its place value. The sum of its values is the value of the whole number. Our number, %10011010, converts to $128+16+8+2 = 154$. Writing it out as a complete algebraic expression looks like this:

$$[(1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (0 * 4) + (1 * 2) + (0 * 1)] = 154$$

The Basic Stamp is a Small Computer

The Basic Stamp is not a very large computer, so its ability to do math is limited. Stamps can't compute numbers over 65535. This is because its internal memory is only 16 bits wide. 2^{16} is 65536, but since the stamp counts the number 0, it is $2^{16}-1 = 65535$. Stamps don't calculate the remainders when dividing. This means that $7/2 = 3$ instead of 3.5. **Stamps always round down with fractions.** We will see how this can be used to our advantage later.

If you need the remainder from a division problem, use a double division sign; $7//2 = 1$ because $7/2$ is 3, remainder 1. Stamps also don't know how to use negative numbers, but we don't need them for the Issac16. Ironically, it is possible to take advantage of the stamps limitations in computing power and turn them to your advantage!

The stamp in the Issac16 is connected to the master processor, and to the lights on the robot's control box. Each **Input** and **Output** to your robot has its own particular name. These names, called **variables**, are short and descriptive.

Before you can use a variable, you need to tell the Stamp:

- 1.) What you want to call it.
- 2.) How large of a value it will be.

This is called **declaring** the variable. Variables come in four sizes: bit, nibble, byte, and word. A bit is the smallest number that the stamp knows about. It is one single binary digit and its value must be either a 1 or a 0. The other variable sizes are shown in the table below.

Name (abbreviation)	Number of Bits	Maximum value
Bit (bit)	1	1
Nibble (nib)	4	15
Byte (byte)	8	255
Word (word)	16	65535

The maximum value of a variable is determined by taking 2 to the power of (Number of Bits) and then subtracting 1 (because 0 is one of the possible values), or,

$$2^{\text{Number of bits}} - 1$$

Note: Assigning variable values of 1 word is costly in terms of available memory resources and it is hardly if ever necessary. Besides, the master processor only accepts data in 1-byte chunks.

Formatting Protocols

Code taken from or referring to the default program, will look be presented in Courier Font, and it will look like this. Commenting in or commenting out is a way to include or omit lines of code. To exclude lines of lines of code or to add comments and explanations of the code, place an apostrophe to the left of the line. Commenting looks like this:

```
`Here is an example of how the default code would look.  
`The default code will go on for more than one line.  
`Default code is blue; any other code will be black.
```

Program File Management

Before reviewing this document, download and print a copy of the IFI Issac 16 Default Code file. This file is available from the IFI website at <http://www.ifirobotics.com/> . The file can be found by clicking on the “Documents” section in the Blue Banner at the top of the IFI home page. The file is called, isaac16-default-002.bsx. This name is very descriptive. It is for the issac16 control box, and is the default program revision 002. The bsx file extension tells you that the type of Basic Stamp that is used in the issac16 is a BS2SX. You can find extra data about the BS2SX over on the manufacturer’s website, [Parallax, Inc.](http://www.parallax.com) When you write a program that you want to keep, save it as issac16-default_with_new_feature-001.bsx. Remember that names can be as long as you want, but should really be limited to about 255 characters. Don’t save over your old working programs when you are trying something new and experimental, use a different revision number at the end of the file name.

Commenting Your Code

Another important item in programming is the **comment**. Whenever the Stamp Editor sees an apostrophe, it ignores everything after that on the same line. You must learn to use this feature to leave notes around the code. *‘Comment your code so you don’t get lost!* Commenting will help you remember what you’ve done or leave a reminder of what you want to do later. Most importantly, comments will still be there when you want to go look at old code that you wrote 6 or more months in the past, and you don’t remember how you did what you did in the old code. Personally, this has saved me from major headaches many times.

Looking at the (issac16-default-002.bsx) Default Program File

The default program starts with a comment block that tells you about where the program came from and has other useful descriptive information. Add whatever you want to the first comment block, and use it as a quick reference when opening a program, to make sure you’ve opened the right one. This is also a good place to leave notes for yourself or other programmers.

```
` PROGRAM:      ISSAC 16 Default Code with One or Two Joystick Control Rev 002  
` Written by:  Innovation First, Inc.  
` Date:        20 SEP 01  
`  
` Define BS2-SX Project Files  
`  
` {$STAMP BS2SX}
```

This second comment block is a header to the variable section of the default program. These are the basic variables used by the program to perform its functions.

```
'===== DECLARE VARIABLES =====  
'  
' Below is a list of declared input and output variables. Comment or un-comment  
' the variables as needed. Declare any additional variables required in  
' your main program loop. Note that you may only use 26 total variables.'
```

Resource Limitations are Everywhere

The design premise underlying the entire BattleBots IQ Engineering/Design methodology is that BattleBot designers and builders are constrained by resources. We call these the 5 budgets: Time, Money, Knowledge, Weight, and Power. You never have enough of any of these resources. The robot that results from your efforts can be thought of as the summation of the compromises that resulted from the algebraic summation of this expression:

$$\text{Time} + \text{Money} + \text{Knowledge} + \text{Weight} + \text{Power} = \text{BattleBot}$$

On Board Processing power is a finite resource, as is the knowledge required to maximize this limited resource. USE IT WISELY.

The Variable Hotel Has Only 26 Rooms

The Stamp is a very small micro-controller, so it only has 26 bytes of storage space for variables. Since each variable is one byte long in the default program, you can only use 26 of them. Any variables you declare for your own use must be included in the 26 bytes.

The default variables are:

```
'----- Operator Interface (OI) - Analog Inputs -----  
p1_x    VAR byte    'Port 1, X-axis on Joystick  
p2_x    VAR byte    'Port 2, X-axis on Joystick  
p3_x    VAR byte    'Port 3, X-axis on Joystick  
p4_x    VAR byte    'Port 4, X-axis on Joystick'
```

These four variables store the information from the X-axis (left-right) of each joystick. Each joystick position has a value from 0 to 255. We have illustrated this “Variable to Joystick” relationship clearly in Control Lesson 2, “Control Basics: Loading and understanding the Default Program”. We will continue to talk more about joysticks, and why the highest value is 255. Note: p1 means port #1 on the operator’s control box.

```
p1_y    VAR byte    'Port 1, Y-axis on Joystick  
p2_y    VAR byte    'Port 2, Y-axis on Joystick  
p3_y    VAR byte    'Port 3, Y-axis on Joystick  
p4_y    VAR byte    'Port 4, Y-axis on Joystick'
```

The variable p1_y is the variable assigned to the Y-axis of the joystick plugged into Port #1.

```
p1_wheel VAR byte    'Port 1, Wheel on Joystick  
p2_wheel VAR byte    'Port 2, Wheel on Joystick  
p3_wheel VAR byte    'Port 3, Wheel on Joystick  
p4_wheel VAR byte    'Port 4, Wheel on Joystick'
```

p1_wheel is the thumbwheel of the joystick connected to Port #1.

```
'p1_aux VAR byte      'Port 1, Aux on Joystick
'p2_aux VAR byte      'Port 2, Aux on Joystick
'p3_aux VAR byte      'Port 3, Aux on Joystick
'p4_aux VAR byte      'Port 4, Aux on Joystick
```

p1_aux is the auxiliary analog input of Port #1. The standard (default) joystick does not have a physical control to use with this variable, and most people don't use the aux input. Since it's not used very often, it is commented out. It is important to understand that p1_aux is available to designers as an ADDITIONAL operator input. The Wires, the variable and the memory space exist so that BattleBot designers can add more operator control switches, joysticks and sensors. Custom controllers will be dealt with in a later (advanced programming) section.

```
PWM1    VAR byte      'Modified output for PWM1      <----ADDED
PWM2    VAR byte      'Modified output for PWM2      <----ADDED
```

These variables are an important part of the default Issac16 code. The variables PWM1 and PWM2 will be used later in the program to change the control method of the robot from 2 stick, "Tank Style" control, to one Stick control. To better "See" how these variables are used, refer to section 4 of Control Lesson 2, "Control Basics: Loading and understanding the Default Program".

```
'----- Operator Interface - Digital Inputs -----
oi_swA  VAR byte      'OI Digital Switch Inputs 1 thru 8
oi_swB  VAR byte      'OI Digital Switch Inputs 9 thru 16
```

If You Only Need One Bit, Then Why Take A Byte?

The oi_swA and oi_swB variables hold the information about the digital inputs from the controller. oi_sw stands for "Operator Interface, Switch inputs." Since a button only has two physical states, off and on, it only requires one bit to store the data about the position of the switch. When the bit holds a 1, then the switch must be on, a 0 means that the switch is off. You might be catching on enough to question why a byte's worth of memory has been added to a variable requiring only a bit's worth of resources?

A byte variable, you might remember, holds 8 bits of information, so each of the switch variables is holding the information for 8 switches. You'll see how to extract that information from the variable a little later on.

The next section is mostly commented out, and includes the analog sensors on your robot. Analog sensors like potentiometers can give input back to your program. The last sensor, bat_volt is hardwired to the battery on your robot, so you could have your program respond to low battery voltages.

```
'----- Robot Controller (RC) - Analog Inputs -----
'sensor1 VAR byte      'RC Analog Input 1, connector pin 2
'sensor2 VAR byte      'RC Analog Input 2, connector pin 16
'sensor3 VAR byte      'RC Analog Input 3, connector pin 5
```

```
'sensor4 VAR byte      'RC Analog Input 4, connector pin 19
'sensor5 VAR byte      'RC Analog Input 5, connector pin 8
'sensor6 VAR byte      'RC Analog Input 6, connector pin 22
'sensor7 VAR byte      'RC Analog Input 7, connector pin 11
'bat_voltVAR byte      'RC Analog Input 8, hardwired to the Battery
                        'Vin = ((4.7/14.7)* Battery voltage)-0.4
                        'Binary Battery Voltage = (Vin/5.0 V)*255
```

Next are the digital switch inputs on your robot. Remember, there are two ways for the Isaac Control Systems to obtain information (**Inputs**):

1.) Operator INPUTS.

The operator can control joystick, and switches that can be modified by code to control the robot

2.) On Board INPUTS. Switches and sensors on board the robot can be configured to monitor and manage mechanical systems. This gives the Robot the ability to react to information autonomously and in real time.

Just like the switch inputs from the Operator Controller, each Robot Controller switch gets one bit in a byte-sized variable. The Isaac16 has only ½ the rc inputs of the Isaac 32. The Isaac 16 doesn't have connections available for switches 9 through 16, so the rc_swB variable is not necessary.

The Robot Controller can manage two types of Outputs:

- 1.) Digital or 2 state outputs.
- 2.) PWM Outputs.

It is important to note that (PWM) Pulse Width Modulated Outputs are a specific type of analog output designed to operate speed controllers and servos.

Speed controllers and servos are controlled by sending them “Pulses” or data bits (0's or 1's) for a specified period of time. This period of time is referred to as the pulse width. Typical servos are controlled (full forward to full backward) by a pulse width range between 1 and 2 milliseconds.

What are Milliseconds?

1 (ms) milliseconds, is 1/1000th of a second. 1 (µs) is 1/1,000,000th of a second. Simple division -

$$1,000,000/1,000 = 1,000$$

- indicates that there are 1000 (µs) microseconds in a (ms) millisecond. Continuing the logic, we find there are 1000 (ms) milliseconds in a second. This is important to the Isaac 16 programmer only as nice to know information since the Isaac Robot controllers do not provide users with direct access to the stamp via the complete set of PBasic instructions. This means the user cannot change the pulse widths. The Isaac Master Processor controls the PWM values.

```
'----- Robot Controller - Digital Inputs -----
rc_swA  VAR byte      'RC Digital Inputs 1 thru 8
rc_swB  VAR byte      'RC Digital Inputs 9 thru 16
```

All of the digital outputs on your robot (things like the Spike relay motor controller) are connected to relay ports clearly marked on the Robot Controller PC Board. The `RelayA` and `RelayB` variables contain 2 bits for every relay. The first bit tells the relay to go forward, the second bit tells the relay to go reverse.

```

'----- Robot Controller - Digital Outputs -----
relayA  VAR byte
relayB  VAR byte

'----- Misc. -----
packet_num  VAR byte
'delta_t VAR byte

```

Notice that a lot of these have been commented out. With the number of possible inputs and outputs for your robot, you can easily run out of variable space. Remember, it's wise to preserve the memory only for variables you will be using.

“Her name was McGill, she called herself Lil’ and everyone knew her as Nancy....”

An alias is just another name for the same thing. A good analogy might be nicknames. Someone (a variable) might be called `Robert`. In American culture, a common alias for `Robert` is `Bob`. Whenever we talk about either `Robert` or `Bob`, we mean `Robert`. In code, this would look like

```

Robert  VAR byte
Bob      VAR Robert      'Bob is an alias for Robert.

```

Aliases don't use any extra space in memory. When you download a program to the Issac16, the Program Editor does a little bit of pre-processing. One process it performs is the replacement of any aliases with the real name of the original variable. So when we download the code containing `Bob`, the program editor will replace the word `Bob` with `Robert`, every time it finds `Bob` in the code.

Aliases can also refer to a part of another variable. To name the part of the original variable, use the name of the variable, a period, and the name of the part. If each of the bits in the byte called `Robert` is labeled 0 through 7, and (say) bit #3 has something to do with `Robert's` left hand, we could talk about that bit all by itself using

```

Left_hand  VAR Robert.bit3  'Also talks about Bob.bit3, because we
                        'already defined that Bob is Robert.

```

The next chart will show you the parts of a byte and what they are called. Let's assume that the name of the variable is `Bob`, to keep it short. This chart applies to all byte variables and aliases. It is sometimes useful to have an alias of an alias. Aliases can be created indefinitely; as long as the item you are creating an alias of already exists.

Bit #	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1
Nibbles	Bob.highnib				Bob.lownib			
	Bob.nib1				Bob.nib0			

Bits	Bob.bit7	Bob.bit6	Bob.bit5	Bob.bit4	Bob.bit3	Bob.bit2	Bob.bit1	Bob.bit0
	Bob.highbit							Bob.lowbit

The entire right half of the chart also applies to all nibble sized variables. The pattern is easy to follow to see how the chart applies to word sized variables.

In our control program, we have already declared the variables oi_swA and oi_swB. You'll remember that each bit in these variables is holding the data having to do with a particular input switch. We can use aliases to give each bit a name so that we don't have to remember which bit does what. The next listing shows all of the aliases defined in the Issac16 default code. When the code is talking about having the inputs duplicated in the other ports, it means that you can connect extra switches to the other ports and they will have the same effect as pressing the buttons listed in the code, such as the joystick trigger.

```

===== DEFINE ALIASES =====
Aliases are variables which are sub-divisions of variables defined
above. Aliases don't require any additional RAM.

----- Aliases for each OI switch input -----
Below are aliases for the digital inputs located on the Operator Interface.
Ports 1 & 3 have their inputs duplicated in ports 4 & 2 respectively. The
inputs from ports 1 & 3 may be disabled via the 'Disable' dip switch
located on the Operator Interface. See Users Manual for details.

p1_sw_trig    VAR oi_swA.bit0    'Joystick Trigger Button,    same as Port4 pin5
p1_sw_top     VAR oi_swA.bit1    'Joystick Top Button,       same as Port4 pin8
p1_sw_aux1    VAR oi_swA.bit2    'Aux input,                 same as Port4 pin9
p1_sw_aux2    VAR oi_swA.bit3    'Aux input,                 same as Port4 pin15

p3_sw_trig    VAR oi_swA.bit4    'Joystick Trigger Button,    same as Port2 pin5
p3_sw_top     VAR oi_swA.bit5    'Joystick Top Button,       same as Port2 pin8
p3_sw_aux1    VAR oi_swA.bit6    'Aux input,                 same as Port2 pin9
p3_sw_aux2    VAR oi_swA.bit7    'Aux input,                 same as Port2 pin15

p2_sw_trig    VAR oi_swB.bit0    'Joystick Trigger Button
p2_sw_top     VAR oi_swB.bit1    'Joystick Top Button
p2_sw_aux1    VAR oi_swB.bit2    'Aux input
p2_sw_aux2    VAR oi_swB.bit3    'Aux input

p4_sw_trig    VAR oi_swB.bit4    'Joystick Trigger Button
p4_sw_top     VAR oi_swB.bit5    'Joystick Top Button
p4_sw_aux1    VAR oi_swB.bit6    'Aux input
p4_sw_aux2    VAR oi_swB.bit7    'Aux input

```

Your robot's control board also has a bank of switch inputs, so the code must provide aliases for these as well. The default program provides for 16 digital inputs, but you only have connections for 8 switches on your robot.

```

----- Aliases for each RC switch input -----
Below are aliases for the digital inputs located on the Robot Controller.

rc_sw1  VAR rc_swA.bit0
rc_sw2  VAR rc_swA.bit1
rc_sw3  VAR rc_swA.bit2
rc_sw4  VAR rc_swA.bit3
rc_sw5  VAR rc_swA.bit4
rc_sw6  VAR rc_swA.bit5

```

```
rc_sw7   VAR rc_swA.bit6
rc_sw8   VAR rc_swA.bit7
rc_sw9   VAR rc_swB.bit0
rc_sw10  VAR rc_swB.bit1
rc_sw11  VAR rc_swB.bit2
rc_sw12  VAR rc_swB.bit3
rc_sw13  VAR rc_swB.bit4
rc_sw14  VAR rc_swB.bit5
rc_sw15  VAR rc_swB.bit6
rc_sw16  VAR rc_swB.bit7
```

Similarly, each bit in the variables RelayA and RelayB controls the direction of each relay output on your robot. Another way to say this is to say that the bits are *mapped* to the outputs. The relay outputs are designed so that if you tell them to go both forward and reverse at the same time, it won't go in either direction.

```
'----- Aliases for each RC Relay outputs -----'
' Below are aliases for the relay outputs located on the Robot Controller.

relay1_fwd   VAR RelayA.bit0
relay1_rev   VAR RelayA.bit1
relay2_fwd   VAR RelayA.bit2
relay2_rev   VAR RelayA.bit3
relay3_fwd   VAR RelayA.bit4
relay3_rev   VAR RelayA.bit5
relay4_fwd   VAR RelayA.bit6
relay4_rev   VAR RelayA.bit7

relay5_fwd   VAR RelayB.bit0
relay5_rev   VAR RelayB.bit1
relay6_fwd   VAR RelayB.bit2
relay6_rev   VAR RelayB.bit3
relay7_fwd   VAR RelayB.bit4
relay7_rev   VAR RelayB.bit5
relay8_fwd   VAR RelayB.bit6
relay8_rev   VAR RelayB.bit7
```

Some Things Never Change

Unlike a variable, a **constant** is a value that will **never change**. When you define a constant in the Issac16, it is just an alias for a specified number or value. When you download your code to the Issac16, the preprocessing program will replace the constants with the value listed. The first 32 constants in the default program tell the master processor which variables you will be using. Remember! You can only choose 26 of these 32 constants, or less. Replacing names with constants is another thing the preprocessor does when the program is downloaded to the Issac16.

```
'===== DEFINE CONSTANTS FOR INITIALIZATION ====='
'=====
' The initialization code is used to select the input data used by PBASIC.
' The Master micro-processor (uP) sends the data you select to the BS2SX
' PBASIC uP. You may select up to 26 constants, corresponding
' to 26 variables, from the 32 available to you. Make sure that you have
' variables for all the bytes received in the serin command.
'
' The constants below have a "c_" prefix, as compared to the variables that
' they will represent.
'
' Set the Constants below to 1 for each data byte you want to receive.
```

' Set the Constants below to 0 for the unneeded data bytes.

Initialize This

If you uncomment any of the variables above, you need to change its partner constant to 1. Conversely, if you comment one or more variables out, to save space, you must change its partner constant to 0. The constants are sent to the master processor when the robot is first turned on so the master processor knows which variables the stamp is looking for.

```
'----- Set the Initialization constants you want to read -----  
c_p1_y          CON    1  
c_p2_y          CON    1  
c_p3_y          CON    1  
c_p4_y          CON    1  
  
c_p1_x          CON    1  
c_p2_x          CON    1  
c_p3_x          CON    1  
c_p4_x          CON    1  
  
c_p1_wheel     CON    1  
c_p2_wheel     CON    1  
c_p3_wheel     CON    1  
c_p4_wheel     CON    1  
  
c_p1_aux       CON    0  
c_p2_aux       CON    0  
c_p3_aux       CON    0  
c_p4_aux       CON    0  
  
c_oi_swA       CON    1  
c_oi_swB       CON    1  
  
c_sensor1      CON    0  
c_sensor2      CON    0  
c_sensor3      CON    0  
c_sensor4      CON    0  
c_sensor5      CON    0  
c_sensor6      CON    0  
c_sensor7      CON    0  
c_batt_volt    CON    0  
  
c_rc_swA       CON    1  
c_rc_swB       CON    1  
  
c_delta_t      CON    0  
c_PB_mode      CON    0  
c_packet_num   CON    1  
c_res01        CON    0
```

Both the robot and the controller have a light on them that indicate low battery voltage. The Issac16 can run on as little as 6 Volts, but larger robots usually require significantly more voltage for their motors. When the robot starts slowing down, it's time to charge the battery again. You can set the voltage at which the low battery light flashes using the formula given in the code. A quick reference chart is in the next table

```

'----- Initialization Constant VOLTAGE - USER DEFINED -----
' This is the 'Low Battery' detect voltage. The 'Low Battery' LED will
' blink when the voltage drops below this value.
' Basically, the value = ((DESIRED FLASH VOLTAGE * 16.3) - 15.2)
' Example, for a 10 Volt Flash trigger, set value = 148.

```

```
dataInitVolt CON 132 '9.0 Volts Flash Trigger
```

Voltage	11.5	11	10.5	10	9.5	9	8.5	8	7.5	7	6.5
Constant	172	164	156	148	140	132	123	115	107	99	91

You Can't Touch This!

The next constants are used to choose the speed at which the Stamp talks to the master processor and should never be changed. The other constants in this section define which input/output pins are being used to connect the processors together and also should not be changed.

```

'===== DEFINE CONSTANTS (DO NOT CHANGE) =====
'=====
' Baud rate for communications with User CPU
OUTBAUD CON 20 ' (62500, 8N1, Noninverted)
INBAUD CON 20 ' (62500, 8N1, Noninverted)

USERCPU CON 4
FPIN CON 1
COMA CON 1
COMB CON 2
COMC CON 3

```

First Things First

Now that we have declared all of the constants and variables we need for the program, we can start to use them in the main program. First we need to *initialize*, or prepare, the communications to the master processor, and all of the output lights.

```

'===== MAIN PROGRAM =====
'=====

```

```

'----- Input & Output Declarations -----

```

```

Output COMB
Input COMA
Input COMC

```

```

Output 7 'define Basic Run LED on RC => out7

Output 8 'define Robot Feedback LED => out8 => PWM1 Green
Output 9 'define Robot Feedback LED => out9 => PWM1 Red
Output 10 'define Robot Feedback LED => out10 => PWM2 Green
Output 11 'define Robot Feedback LED => out11 => PWM2 Red
Output 12 'define Robot Feedback LED => out12 => Relay1 Red

```

```

Output      13      'define Robot Feedback LED => out13 => Relay1 Green
Output      14      'define Robot Feedback LED => out14 => Relay2 Red
Output      15      'define Robot Feedback LED => out15 => Relay2 Green

'----- Initialize Inputs & Outputs -----

Out7  = 1      'Basic Run LED on RC
Out8  = 0      'PWM1 LED - Green
Out9  = 0      'PWM1 LED - Red
Out10 = 0      'PWM2 LED - Green
Out11 = 0      'PWM2 LED - Red
Out12 = 0      'Relay1 LED - Red
Out13 = 0      'Relay1 LED - Green
Out14 = 0      'Relay2 LED - Red
Out15 = 0      'Relay2 LED - Green

```

Be a Safe Programmer !

For safety, all of the outputs to motors should be initialized to be off. All of the inputs and outputs are contained in byte sized variables, so the values all range between 0 and 255. 0 is reverse and 255 is forward, so halfway between, or 127, stops the motor.

```

p1_x = 127      'Port 1, X-axis on Joystick
p2_x = 127      'Port 2, X-axis on Joystick
p3_x = 127      'Port 3, X-axis on Joystick
p4_x = 127      'Port 4, X-axis on Joystick

p1_y = 127      'Port 1, Y-axis on Joystick
p2_y = 127      'Port 2, Y-axis on Joystick
p3_y = 127      'Port 3, Y-axis on Joystick
p4_y = 127      'Port 4, Y-axis on Joystick

p1_wheel = 127  'Port 1, Wheel on Joystick
p2_wheel = 127  'Port 2, Wheel on Joystick
p3_wheel = 127  'Port 3, Wheel on Joystick
p4_wheel = 127  'Port 4, Wheel on Joystick

'p1_aux = 127   'Port 1, Aux Analog
'p2_aux = 127   'Port 2, Aux Analog
'p3_aux = 127   'Port 3, Aux Analog
'p4_aux = 127   'Port 4, Aux Analog

```

Respect the Master CPU

Be sure to uncomment the aux analog initializations if you use those inputs and variables. Otherwise leave them commented out.

The master initialization routine sends the information about which variables you are using to the master processor. The << symbol means shift left. %01101101 << 1 = %11011010. The constants here take the data from the constants before and assemble them into bytes to be sent to the master processor. It is important that this block of code is not changed because the master processor expects this exact configuration of constants to be sent to it. An error in this section would cause the master processor to send the wrong variables back to your program. You wouldn't want a pneumatic cylinder

firing when you expected a motor to run. Always be careful when using your robot with a brand new program in it. It may do unexpected things if you have a problem in your code.

```

===== PBASIC - MASTER uP INITIALIZATION ROUTINE =====
=====
' DO NOT CHANGE THIS! DO NOT MOVE THIS!
' The init routine sends 5 bytes to the Master uP, defining which data bytes to receive.
' 1) Collect init.
' 2) Lower the COMA line, which is the clk line for the shift out command.
' 3) Lower COMB line to tell pic that we are ready to send init data.
' 4) Wait for pic to lower the COMC line, signaling pic is ready for data.
' 5) Now send out init data to pic, all 5 bytes.
' 6) Now set direction and levels for the COMA and COMB pins.

tempA          CON   c_p3_x          << 1 + c_p4_x << 1 + c_p1_x << 1 + c_p2_x << 1 + c_rc_swB
dataInitA      CON   tempA           << 1 + c_rc_swA << 1 + c_oi_swB << 1 + c_oi_swA
tempB          CON   c_sensor4 << 1 + c_sensor3 << 1 + c_p1_y << 1 + c_p2_y << 1 + c_sensor2
dataInitB      CON   tempB           << 1 + c_sensor1 << 1 + c_packet_num << 1 + c_PB_mode
tempC          CON   c_batt_volt << 1 + c_sensor7 << 1 + c_p1_wheel << 1 + c_p2_wheel << 1 + c_sensor6
dataInitC      CON   tempC << 1 + c_sensor5 << 1 + c_p3_y << 1 + c_p4_y
tempD          CON   c_res01 << 1 + c_delta_t << 1 + c_p3_aux << 1 + c_p4_aux << 1 + c_p1_aux
dataInitD      CON   tempD << 1 + c_p2_aux << 1 + c_p3_wheel << 1 + c_p4_wheel

```

This shiftout command does the actual sending of initialization data to the master processor.

```

Output  COMA
low     COMA
low     COMB
Wait_init:  if IN3 = 1 then Wait_init:
Shiftout COMB,COMA,1, [dataInitA,dataInitB,dataInitC,dataInitD,dataInitVolt]
Input   COMA
high    COMB

```

The first statement in the loop is a *label*, called `MainLoop`. This is where the loop will start again when we `goto MainLoop` at the end of the loop.

```

===== MAIN LOOP =====
=====
MainLoop:

```

The `Serin` command gets all of the input data from the master processor. Since we've already sent it information about which inputs we want, we need to make a `Serin` command that receives those inputs in the order sent from the master processor and excludes any of the variables we didn't define above.

```

----- Serin Command - Get Data from Master uP -----
' Construct the "serin" command using the following rules:
' 1) There must be one variable for every input defined in the "Define Constants for Init"
section.
' 2) The order must match the order in the EXAMPLE SERIN COMMAND below.
' 3) The total number of all variables may not exceed 26.
' 4) Only use one "Serin" command.
' 5) The Serin command must occupy one line.

```

If you see a BASIC INIT ERR on the Robot Controller after programming and pressing RESET, then there is a problem with the Serin command below. Check the number of variables. A BASIC INIT ERR will not occur if you have the variables in the wrong order, however, your code will not work correctly.

EXAMPLE SERIN COMMAND

This example exceed the 26 variable limit and is not on one line:

```
Serin COMA\COMB, INBAUD,  
[oi_swA,oi_swB,rc_swA,rc_swB,p2_x,p1_x,p4_x,p3_x,PB_mode,packet_num,sensor1,  
sensor2,p2_y,p1_y,sensor3,sensor4,p4_y,p3_y,sensor5,sensor6,p2_wheel,p1_wheel,  
sensor7,sensor8,p4_wheel,p3_wheel,p2_aux,p1_aux,p4_aux,p3_aux,delta_t,res01]
```

The default Serin construction is on the next line. Note that it must be entirely on one line and not split as it is shown here.

```
Serin COMA\COMB, INBAUD,  
[oi_swA,oi_swB,rc_swA,rc_swB,p2_x,p1_x,p4_x,p3_x,packet_num,p2_y,p1_y,p4_y,p3_y,p2_wheel,p1_wheel,p  
4_wheel,p3_wheel]
```

Know Your Speed

This Toggle command blinks the “Basic Run” light every two loops; one loop on, one loop off. If your program seems to be running slowly, you can use this light as an indicator to the actual speed your program is running. Stamps do not run at a specific speed. Some commands take longer to execute than others. The stamp simply executes code as quickly as it can. The slowest command is the debug command. Only use debug for testing, and comment out the debug commands before you download the program for the last time.

```
'----- Blink BASIC RUN LED -----  
Toggle 7      'Basic Run LED on the RC is toggled ON/OFF every loop.
```

Do the Math

Here is the part of the default program that takes all of your inputs and translates them to outputs. All math to be performed on the inputs before sending them to the master processor is done here.

```
'===== PERFORM OPERATIONS =====  
' Add your custom code here.  
' Delete any of the following sections below (except for Output Data) as desired.
```

The Zen of Pbasic

The first line, `relay1_fwd = p1_sw_trig &~ rc_sw1`, takes the input from the trigger on the first joystick and makes relay1 go forward *unless* the switch connected to input1 on the robot is *closed*. When a switch is *closed*, the two wires inside of it are connected together, or are *shorted* together.

If you wanted something to happen automatically when a switch was hit, you could write, `output_name = input1 & input2`. You would have to replace `output_name` with the relay direction you want to control, such as `relay4_fwd`, and

input1 and input2 with the inputs you want to cause the action. You can add as many controls as you want to an output by *ANDing* them together. Another example; if you want relay3 to go forward when the trigger on any joystick is pressed, the code would be

```
relay3_fwd = p1_sw_trig & p2_sw_trig & p3_sw_trig & p4_sw_trig.
```

```
'----- Buttons to Relays -----'
' The & used below is the PBASIC symbol for AND
' The &~ used below is the PBASIC symbol for AND NOT

relay1_fwd = p1_sw_trig &~ rc_sw1      'Port 1 Trigger = Relay 1 Forward
                                       'Relay 1 won't go Forward if rc_sw1 is ON

relay1_rev = p1_sw_top &~ rc_sw2      'Port 1 Thumb = Relay 1 Reverse
                                       'Relay 1 won't go Reverse if rc_sw2 is ON

relay2_fwd = p2_sw_trig &~ rc_sw3     'Port 2 Trigger = Relay 2 Forward
                                       'Relay 2 won't go Forward if rc_sw3 is ON

relay2_rev = p2_sw_top &~ rc_sw4     'Port 2 Thumb = Relay 2 Reverse
                                       'Relay 2 won't go Reverse if rc_sw4 is ON

relay3_fwd = p3_sw_trig              'Port 3 Trigger = Relay 3 Forward
relay3_rev = p3_sw_top                'Port 3 Thumb = Relay 3 Reverse
relay4_fwd = p4_sw_trig              'Port 4 Trigger = Relay 4 Forward
relay4_rev = p4_sw_top                'Port 4 Thumb = Relay 4 Reverse

relay5_fwd = p1_sw_aux1              'Port 1 Aux1 = Relay 5 Forward
relay5_rev = p1_sw_aux2              'Port 1 Aux2 = Relay 5 Forward
relay6_fwd = p3_sw_aux1              'Port 3 Aux1 = Relay 6 Forward
relay6_rev = p3_sw_aux2              'Port 3 Aux2 = Relay 6 Forward
relay7_fwd = p4_sw_aux1              'Port 4 Aux1 = Relay 7 Forward
relay7_rev = p4_sw_aux2              'Port 4 Aux2 = Relay 7 Reverse
relay8_fwd = 1                       'Relay 8 always Forward
relay8_rev = 0                       'Relay 8 always Forward
```

Relay8 is set to always be forward because this code is adapted from the code for the USFIRST robotics competition, where each robot is required to have a relay-run light that is always on during the time when the robot is running. Your control box doesn't have a connection for relays 5 through 8, so this code won't do anything on your robot.

These commented lines turn the green and red (forward and reverse, respectively) lights on when the value of the y-axis of each joystick is greater than 216 (green, forward) or less than 56 (red, reverse).

```
'----- Feedback LEDs for PWM1, PWM2 -----'
'Out8 = p1_y/216      'LED is ON when Victor883 full forward <---REMOVED
'Out9 = ~(p1_y/56 max 1) 'LED is ON when Victor883 full reverse <---REMOVED
'Out10 = p2_y/216     'LED is ON when Victor883 full forward <---REMOVED
'Out11 = ~(p2_y/56 max 1) 'LED is ON when Victor883 full reverse <---REMOVED

'----- Feedback LEDs for Relay1, Relay2 -----'
```

This is an example of taking advantage of the stamps computing limitations. If p1_y's value is ever below 216, then p1_y/216 will equal 0.something (Read: "0 point something"). It doesn't matter what the something is, because the stamp rounds down to whole numbers. Remember that the stamp does not use fractions or decimal places. Any number

above 216 will give the result of 1.something. Again, the something doesn't matter, it gets cut off, leaving only a 1. That 1 is used to turn on the green light. If the number were ever to reach 432 (2 * 216), our output would be 2. Since the maximum value of p1_y is 255 (because it is only a byte long), a 432 could *never* happen.

The red lights use $\sim(p1_y/56 \text{ max } 1)$. \sim means NOT, which changes all ones to zeros and vice versa. Since p1_y/56 is limited to be no greater than 1 (by the MAX 1 command), p1_y/56 will always be either a 0 or 1. When you NOT a 0 or 1, you get 1 or 0. This turns on the red light when p1_y is less than 56.

There are four more indicators as to what your robot is doing with its first two relays. Again, green is forward and red is reverse.

```
Out13 = relay1_fwd      'LED is ON when Relay 1 is Forward
Out12 = relay1_rev      'LED is ON when Relay 1 is Reverse
Out15 = relay2_fwd      'LED is ON when Relay 2 is Forward
Out14 = relay2_rev      'LED is ON when Relay 2 is Reverse
```

Read the Code Below to Understand The Following Stuff

Each if...then statement checks a digital switch input and limits an analog output. As a result, we call the switches connected to these inputs *limit switches*. Switch 5, when shorted, keeps PWM3 (analog output 3) from having a value greater than 127; otherwise it has no effect. The construction of the if...then statement here is a little different than you would expect, so I will dissect it a little bit more and show you two alternative methods of performing the same function.

If you are familiar with any other programming languages, you might have seen an if...then statement before. The common definition of an if...then statement is:

If (part 1) then (part 2). If part 1 is not equal to zero, or if part 1 is true, then do part 2. In other programming languages, part 2 could be either an instruction to perform a function, or an instruction to GOTO somewhere. With a stamp, the only option for part 2 is to tell it to GOTO a label. Stamps if...then statements are always written as

```
| If (something is true) then label.
```

Normally, you might say that "If the limit switch is being pressed (rc_sw = 1) then stop the motor from going forward (don't allow the value of the motor output be greater than 127.)"

To write this requires more code than you might initially think.

```
| if rc_sw5 = 1 then limit_p3_y
goto next1
limit_p3_y:
p3_y = p3_y MAX 127
next1:
```

In English: If the switch is closed, then goto the part of the program that does the limiting. If not, the next line continues by going to the part in the code that skips the limiting. This is perfectly valid, but it takes up a lot of space and time. The way in the default code uses reversed logic - If the switch is *not* closed, then skip the next line, which limits the output.

The most efficient, but harder to understand way is to use a mathematical formula to limit the output.

```
| p3_y = p3_y MAX (255 - (128 * rc_sw5)) MIN (rc_sw6 * 127) 'limit p3_y with switches
```

If rc_sw5 is 0, then the MAX value is $255 - (128 * 0) = 255 - 0 = 255$. If rc_sw5 is 1, then the MAX value of p3_y is $255 - (128 * 1) = 255 - 128 = 127$, therefore limiting the input to 127.

p3_y input	rc_sw5	rc_sw6	MAX	MIN	output
0 - 127	0	0	255	0	0 - 127
0 - 127	0	1	255	127	127
0 - 127	1	0	127	0	0 - 127
0 - 127	1	1	127	127	127
127 - 255	0	0	255	0	127 - 255
127 - 255	0	1	255	127	127 - 255
127 - 255	1	0	127	0	127
127 - 255	1	1	127	127	127

Doing the limiting as a mathematical function is very fast, and in this case replaces two entire if...then statements.

```

'----- PWM outputs Limited by Limit Switches -----
if rc_sw5 = 0 then next1:
    p3_y = p3_y MAX 127
next1:

if rc_sw6 = 0 then next2:
    p3_y = p3_y MIN 127
next2:

if rc_sw7 = 0 then next3:
    p4_y = p4_y MAX 127
next3:

if rc_sw8 = 0 then next4:
    p4_y = p4_y MIN 127
next4:

if rc_sw9 = 0 then next5:
    p1_wheel = p1_wheel MAX 127
next5:

if rc_sw10 = 0 then next6:
    p1_wheel = p1_wheel MIN 127
next6:

if rc_sw11 = 0 then next7:
    p2_wheel = p2_wheel MAX 127
next7:

if rc_sw12 = 0 then next8:
    p2_wheel = p2_wheel MIN 127
next8:

if rc_sw13 = 0 then next9:
    p3_wheel = p3_wheel MAX 127
next9:

if rc_sw14 = 0 then next10:
    p3_wheel = p3_wheel MIN 127
next10:

```

```

if rc_sw15 = 0 then next11:
    p4_wheel = p4_wheel MAX 127
next11:
if rc_sw16 = 0 then next12:
    p4_wheel = p4_wheel MIN 127
next12:

```

The next code block is the code that drives the robot using only one joystick.

Refer to Branden's other document in the, Learning Tools Section, for a detailed description of this coding.

```

'----- One Joystick Drive + Feedback LEDs for PWM1, PWM2 -----
' This section modified the output of PWM1, and PWM 2 for control from one
' joystick (Port 1). The Out8 thru Out11 lines control the Feedback LEDs.
' Y-axis controls speed.
' X-axis turns.
' PWM1 - Left motor.
' PWM2 - Right motor.

PWM1 = (((2000 + p1_y - p1_x + 127) Min 2000 Max 2254) - 2000) '<---ADDED
PWM2 = (((2000 + p1_y + p1_x - 127) Min 2000 Max 2254) - 2000) '<---ADDED

Out8 = pwm1/216 'LED is ON when Victor883 full forward '<---ADDED
Out9 = ~(pwm1/56 max 1) 'LED is ON when Victor883 full reverse '<---ADDED
Out10 = pwm2/216 'LED is ON when Victor883 full forward '<---ADDED
Out11 = ~(pwm2/56 max 1) 'LED is ON when Victor883 full reverse '<---ADDED

```

Now that we've modified all of the data for our robot, we need to send it to the master processor to be used in actually making the robot move. The Serout command sends 255 twice to tell the CPU that it's the beginning of communication. Then it sends the values for each PWM and relay output. Since the default code changes the control style from one joystick to two, they used the `pwm1` and `pwm2` variables from above in the Serout command.

```

'===== OUTPUT DATA =====
'=====
' The Serout line sends data to the Output uP. The Output uP passes this to each PWM 1-16
' and Relay 1-8. The Output uP will not output data if there is no communication with the
' Operator Interface or if the Competition Mode is Disabled. Do not delete any elements
' from the Serout array. Set unused PWM outputs to 127. Set unused relay outputs to 0.
'
' Serout USERCPU, OUTBAUD,
[255,255,(PWM1),relayA,(PWM2),relayB,(PWM3),(PWM4),(PWM5),(PWM6),(PWM7),(PWM8),(PWM9),(PWM10),(PWM1
1),(PWM12),(PWM13),(PWM14),(PWM15),(PWM16)]

```

Again, this next line should be on one line, and not split up as it's shown here.

```

Serout USERCPU, OUTBAUD,
[255,255,pwm1,relayA,pwm2,relayB,p2_y,p3_y,p4_y,p3_x,p4_x,p1_wheel,127,127,127,127,127,127,127,127]
'^---ADDED PWM1,PWM2 TO SEROUT COMMAND ABOVE

```

Next we have to jump back up to the top of the loop to do this all over again.

```
Goto MainLoop:
```

Though it will never get to this point, it is always a good idea to have a Stop command at the end of the code.

| Stop

There is about $\frac{3}{4}$ of the space left in memory for more code, so you can add a lot to this program. Later white papers will discuss how to change the default program for your needs. Experimenting with the code on your own is encouraged, using clues you might have gained from this white paper.

If you're still awake and paying attention...know you are destined to become the team programmer, if it took you thirteen tries to just understand the first three pages, then know you're pretty normal.

Good luck in your programming efforts, and remember...learning to code, like anything else is about DOING IT.