Description

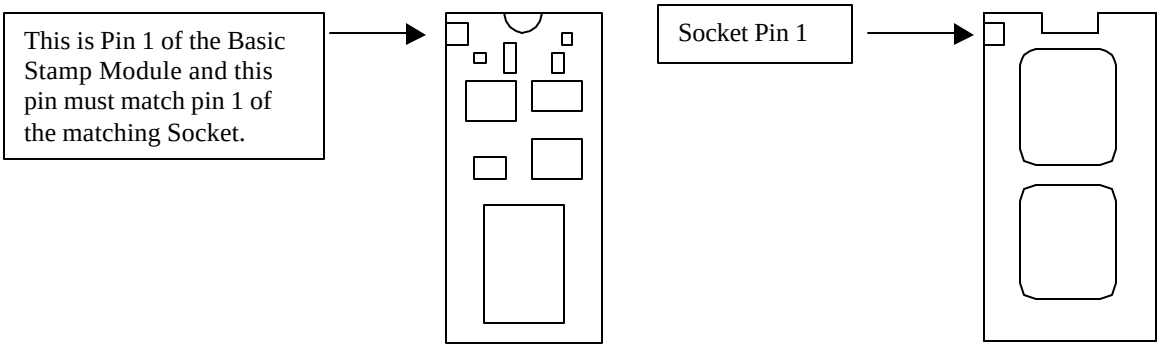
The BSIO16ex module is designed for use in Industrial Machinery as a main controller, or as a controller to add features to a machine. This board is the half-size version of the BSIO32ex Module. Standard 24-volt DC power is used for both for input sensor and output load devices. This board will support the BASIC Stamp BS2p24 and all other 24-pin stamp modules. This board may be used to control Industrial Processes, or equipment functions, without having to learn ladder logic. **Before reading further, review the safety and responsibility section on page 4.**

Features

- 8 High Speed Opto-Isolated Inputs
- Inputs are Configurable for either NPN or PNP Devices
- 8 High Current Sinking Drivers, Using a ULN2803 Driver
- Fly-Back Diodes for Protection from Inductive Spikes on Outputs
- Logic State Indicators for Inputs and Outputs
- 1 LCD Port for use with a Serial Backpack equipped LCD
- 1 Expansion Port for Direct I2C Access
- 1 Expansion port for Direct Auxiliary I/O Pin Access
- DIN Rail Mounting for Ease of Installation
- On Board Switching Regulator for 5 volt Logic Level Power
- Screw Terminal Connections for Ease of Installation
- On Board Programming Com Port, DB9 Connector

Getting Started

To correctly install your Stamp Module, you must identify pin 1 on the 24 pin Stamp module and match this to pin 1 of the 24 pin DIP Socket on the BSIO16ex board prior to insertion. *(See the BASIC Stamp Programming Manual 2.0, Page 26. The Socket Notch on may be smaller than indicated by the drawing.)*



Note: BASIC Stamps, PBasic and all References to BS2 products are Trademarks of Parallax, Inc.

BSIO16ex Power

The BSIO16ex module will typically be powered from a stand-alone 24-volt DC power supply, which is available from electronic parts distributors, like DigiKey, Newark, or Jameco. This 24-volt DC power must be routed through a DIN rail mounted fuse holder.

All power delivered to this I/O board should be fused for safety. Failure to fuse the power supplied to the board could result in damage to the board electronics, if reverse power is connected.

A shorting diode is installed on the board. Should power be applied in a reverse direction, this shorting diode will conduct and blow the fuse you supplied. The current consumption of each individual sensor and all output loads connected to the module should be measured. This will help determine the capacity of the power supply and the size of fuse needed for safe and reliable operation.

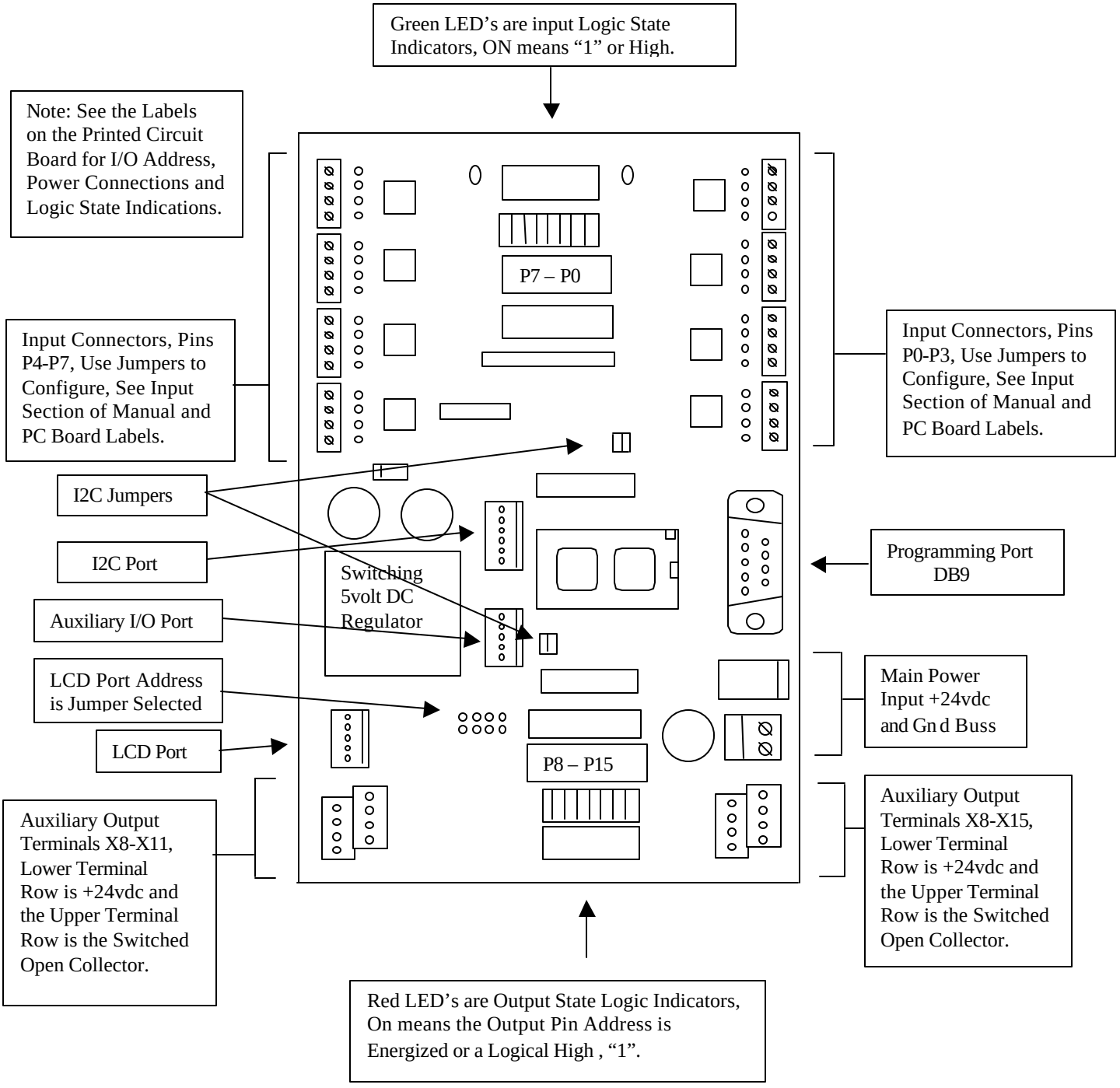
NOTE: The BASIC Stamp's on board 5-volt regulator is not used. All 5-volt DC power for the Stamp is delivered to pin 37 (VDD) on the BS2p40 from the switching regulator (Power Trends) mounted on the BSIO16ex module. *(See the BASIC Stamp Programming Manual 2.0, Pages 16,17.)*

The power supplied to the BSIO16ex module is connected to all input and output terminals via a common power buss. For proper operation, it is very important that no other power source, or additional power supply be used to power loads directly connected to the BSIO16ex module. Use of mismatched power supplies could lead to a false logic state indication on outputs, or damage to electrical components.

A wide range of input voltages maybe applied without affecting the +5vdc logic level power. The switching power supply requires a minimum of +9 volts DC and no more than +30 volts DC to properly condition power. The important consideration is to match the power supplied to your output load devices.

If your application will use or actuate 12-volt devices such as lights, motors, solenoids or relays, then the 13.8 volts DC typical of automotive electrical systems will provide adequate power to the BSIO16ex module to control 12-volt devices.

In the industrial environment most sensors and controls run on 24 volts DC (direct current). Control devices such as relays, pneumatic solenoids and other inductive devices will have a minimum operating voltage, below which, they will not actuate. The minimum operating voltage of the devices you decide to use should be bench tested, using a variable voltage source. For reliable operation you should verify that the voltage supplied to the BSIO16ex module by your power supply will have a positive margin for reliable actuation. Failure to determine your operating margin could result in intermittent operation of loads, as additional loads come on-line.


Green LED's are input Logic State Indicators, ON means "1" or High.
Note: See the Labels on the Printed Circuit Board for I/O Address, Power Connections and Logic State Indications.
P7 – P0
Input Connectors, Pins P4-P7, Use Jumpers to Configure, See Input Section of Manual and PC Board Labels.
Input Connectors, Pins P0-P3, Use Jumpers to Configure, See Input Section of Manual and PC Board Labels.
I2C Jumpers
I2C Port
Auxiliary I/O Port
LCD Port Address is Jumper Selected
LCD Port
Switching 5volt DC Regulator
Programming Port DB9
Main Power Input +24vdc and Gnd Buss
P8 – P15
Auxiliary Output Terminals X8-X11, Lower Terminal Row is +24vdc and the Upper Terminal Row is the Switched Open Collector.
Auxiliary Output Terminals X8-X15, Lower Terminal Row is +24vdc and the Upper Terminal Row is the Switched Open Collector.
Red LED's are Output State Logic Indicators, On means the Output Pin Address is Energized or a Logical High , "1".

Safety and Programming Responsibility Policy

When designing industrial equipment or an industrial process, at no time are the operators of a machine or a process to be exposed to physical dangers created by the process, or machinery. All mechanisms should be appropriately guarded in accordance with OSHA and other industrial standards. Some guards are physical covers others are electronic, such as the use of a light curtain. If you are not familiar with these standards and methods of equipment construction, then you need to do more research, or seek out professional help, prior to implementing any industrial control solution.

Due to the differences and complexity in each application, the responsibility for a safe installation, is in the hands of those implementing the solution. Any program implemented in any PLC must be fully debugged and tested for safe and proper operation. The programming examples contained in this document are for education purposes only, to be used as concepts by the programmer, as a tool to effectively develop and implement an industrial control solution. This document cannot cover all the potential problems, nor can this document address all potential solutions. This is the responsibility of the programmer and those involved with machine design and construction and those engineering an industrial process.

Failure to address issues of safety, machine guards, both physical and electronic, as well as programming to permit the safe interaction of an industrial process with operators may result in injury or death. The purchaser of this product assumes all responsibility for the safe installation and implementation of any industrial process or control solution. Please seek out professional services, if you are not familiar with the requirements and techniques to facilitate a safe installation.

Use of this product constitutes acceptance of this Safety and Programming Responsibility Policy. If you do not agree with this Policy please return product for a refund, less any restocking charges. (This product manual or document is available free from all distributors of this product, to permit the purchaser to review the acceptability of this product, prior to purchase.)

Communications Programming Port

A standard DB9 Connector is provided on the BSIO16ex module to download your program into the Basic Stamp. There is no need to remove the Basic Stamp to program the device. You will need to supply power to the board, as mentioned above, 9-30 volts DC in order to program a Basic Stamp plugged into the board. *(See the Basic Stamp Programming Manual 2.0 for More Details on the Programming Port, Page 27.)*

Outputs

The output connections shown are powered from the main power input terminals. This input power is on a common power buss to all input and output screw terminal locations. The lower terminal row is +VDC and the upper terminal row labeled with address pin numbers 8-15, are a switched Gnd connection. *(See the Printed Circuit Board for these labels)*. These output terminals labeled 8-15 are switched to ground via NPN open collectors from a Power Darlington transistor array.

When the Stamp addresses an output pin, switching high, *(logic level of 1, approx. +5vdc)* one of the Darlington Drivers in the array saturates, sinking power for both the connected load and the Red LED logic

indicator to ground. The red LED lights up to indicate a load at this output is now energized. A diode internal to each transistor of the Darlington array provides transient spike protection, when the transistor is turned off.

NOTE: Do not try to drive a higher voltage device than you have provided power for. Example: Such as trying to drive a 48-volt relay using the 24 volts available at the output terminal. The device will fail to actuate.

NOTE: Do not try to power a high voltage device from an external power supply and then sink this power to the common ground on the BSIO16ex board.

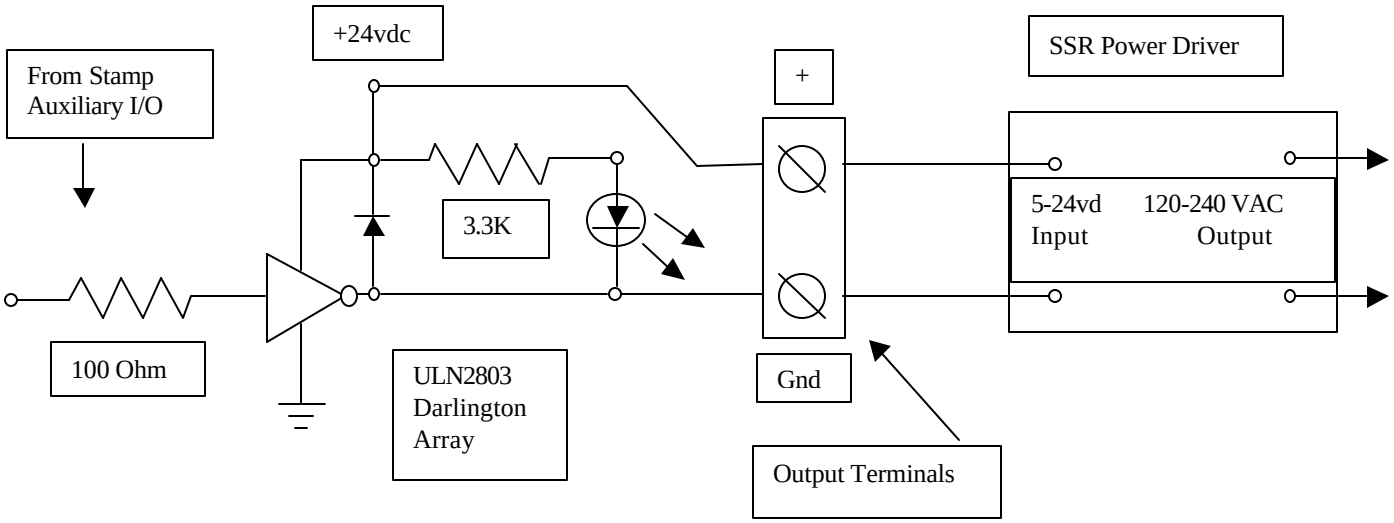
Example: Providing +48 volts to the previously mentioned relay and then sinking this power to a common ground buss will cause faulty operation. The power difference between the 24 and 48 volts will drive current through the flyback diode and may burn out this diode due to excessive current flow. Also once a load is energized, it may fail to disconnect due to this current flow. The current flow from the 48-volt supply may damage the 24-volt supply connected to the BSIO16ex module.

NOTE: Do not try to drive a lower voltage device such as a 12-volt relay using an external 12-volt power supply while supplying 24 volts to the BSIO16ex module. The output LED indicator for that output pin will light up continuously, due to current flow from the 24-volt BSIO power buss through to the 12 volt device to the external power supply.

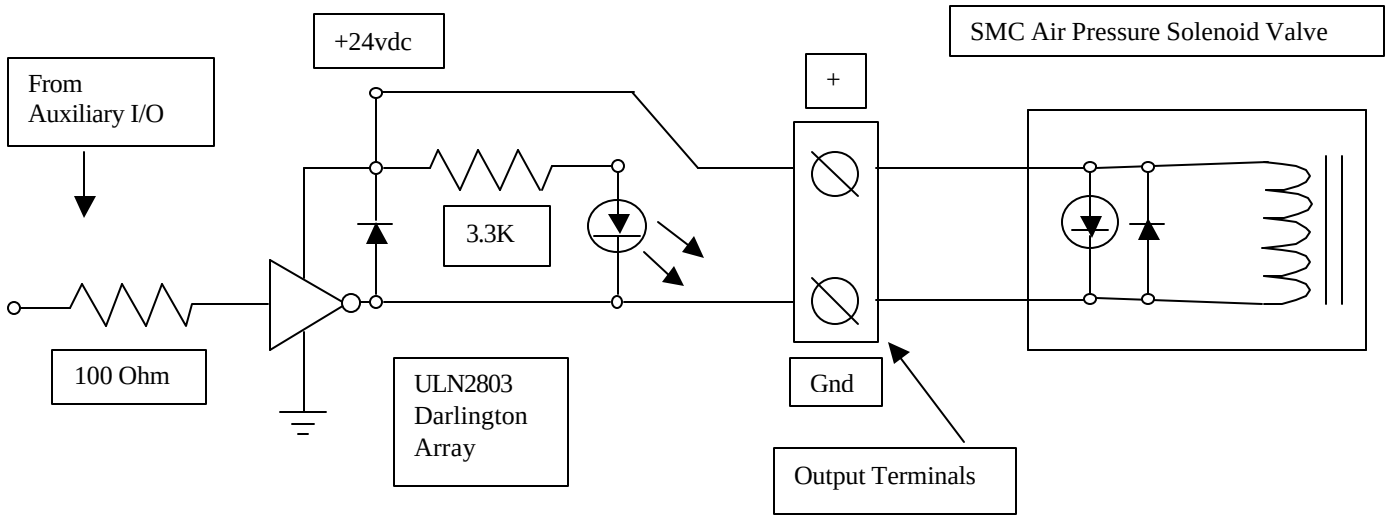
You may power a lower voltage load device using the power supplied to the BSIO16ex module through the output screw terminals, provided, that you install a current-limiting resistor in series with the load. Power supplied to the BSIO16ex module should match the power required *(current and voltage)* for all loads attached to the output terminals. A typical installation would be, 12-14 volts for 12 volt rated loads, 24-26 volts for standard industrial sensors and controls.

Loads like the SMC pneumatic solenoid, may have built in logic indicators and internal fly back diodes, which are polarity sensitive. When connecting to these devices, make sure that the device and the output terminals are wired correct for polarity. The output schematic shown is the entire circuit attached to the output I/O pins. The output circuit is simply duplicated 8 times on the BSIO16ex board.

Typical 24-volt Output Connections



NOTE: All high power loads must be controlled with an appropriate high power driver. The example shown above, using a solid state relay to control a resistive heating element powered by 120-240 VAC is the proper way to control many kilowatts of power using the output pin of a BASIC Stamp.



Note: The red LED's, light up to indicate a load at this output is energized.

Output Pin Control

The BS2 Basic Stamp Module has a total of 16 I/O pins. Pins P8-P15 are used as output pins. The other pins numbered P0-P7 are used as Inputs.

To control the logic state of an output pin, typically one of two commands will be used in PBasic source code software, **High** *Pin#* and **Low** *Pin#.* One way to write a logic state to an output pin is shown below.

(See the Stamp Programming Manual for more details on the commands HIGH and LOW pin#.)

```
                    'Consider a conveyor motor is controlled via a power relay on
                    'Auxiliary I/O Pin # 8.
High          8     'To turn the motor on, use the high pin # command
Low           8     'To turn it off, use the low pin # command
```

As the number of outputs used increases, tracking which I/O pin is connected to what, will become a bit more difficult. Thus to facilitate programming one can use the **Con** *(constant)* assignment ability within the PBasic programming software. The ability to name an output pin with an alias name that identifies the function, makes for easy to read and understandable source code. A good programming technique is to identify these names in the beginning of your program. In this section, you can alias all your output control pin numbers, with specific names, which relate to the functions you are controlling.

```
                              'Output Alias Names
DispenseMotor     con 8       'this is a motor control relay on I/O Pin 11
```

Once you assign a name to a pin number, then you can use this high level name to reference the pin.

```
High  DispenseMotor          'motor turns on

Pause 1000                   'wait one second

Low   DispenseMotor          'motor turns off
```

The sample program listed at the end of this manual, is another example of how alias names may be used in the software coding for easy reading of the outputs being controlled. The words followed by a ":" are labels. Please refer to the Basic Stamp Programming Manual 2.0 for more details on the use of constants, variable names, reserved names for inputs and outputs, program labels and the use of **Goto, Gosub** and **Return** directives in programming. Your application/program complexity may vary, due to the control functions you need to implement.

Input Pin I/O Configuration

Stamp I/O pins P0-P7 are used as inputs. The logic state information will travel one direction on these I/O pins, into the BSIO16ex module. Logic state information cannot be transmitted out on these pins, the exception occurs when using the expansion ports. The input logic state is shown by the green LED array in the center of the board.

The powered input connections as depicted on page 3, are powered from the main power input terminals, as well as the output terminals. One of the four input terminals is labeled +VDC, with the two center terminals being connected to the opto-isolator and the last of the four terminals is labeled Gnd. *(See the labels on the printed circuit board)*.

Note: Where you place the jumper will depend on the flavor, or polarity of the sensor used.

Note: All 8 inputs on the BSIO16ex are normally pulled low. When the green logic state LED for an input pin is illuminated, the LED diode inside the associated opto-isolator is energized. The result is a logical "1" on the Stamp's I/O pin. The output of the opto-isolator sources power from +5 volt buss driving both the driver for the Green LED and the Stamp Input pin high.

When the green LED is not illuminated, the logic state on the stamp I/O pin is a "0", or a LOW condition. When the green LED is illuminated, the logic state on the stamp I/O pin is HIGH condition or a "1".

The input terminals for each input pin are easily configured for a variety of input sensors or control devices. The inputs may be configured to work with either NPN, or PNP output drivers contained in proximity and photo sensors. Switches and open collector transistor sensors can be board powered, or isolated, with their power being supplied from an external source.

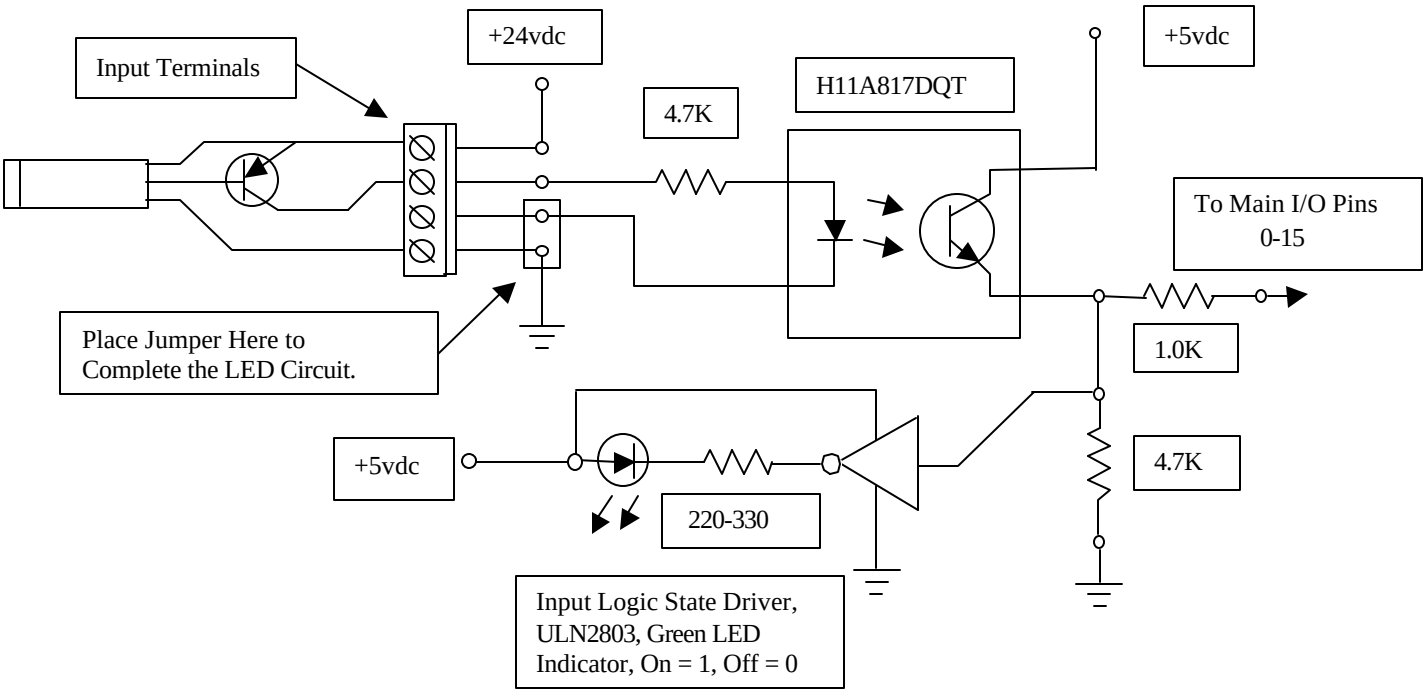
If you are using standard industrial sensors then the following color code will be useful. The black output signal wire is typically terminated on one of the center two terminals depending on the flavor of the sensor, NPN vs. PNP. The whole purpose here is to provide a complete electrical circuit, which will energize the LED within the opto-isolator.

Note: To verify the manufacturer of your sensor is using the same color code for electrical connections, consult the data sheet for your sensor. Most 24-volt industrial proximity or optical sensors will follow the wiring color code listed below.

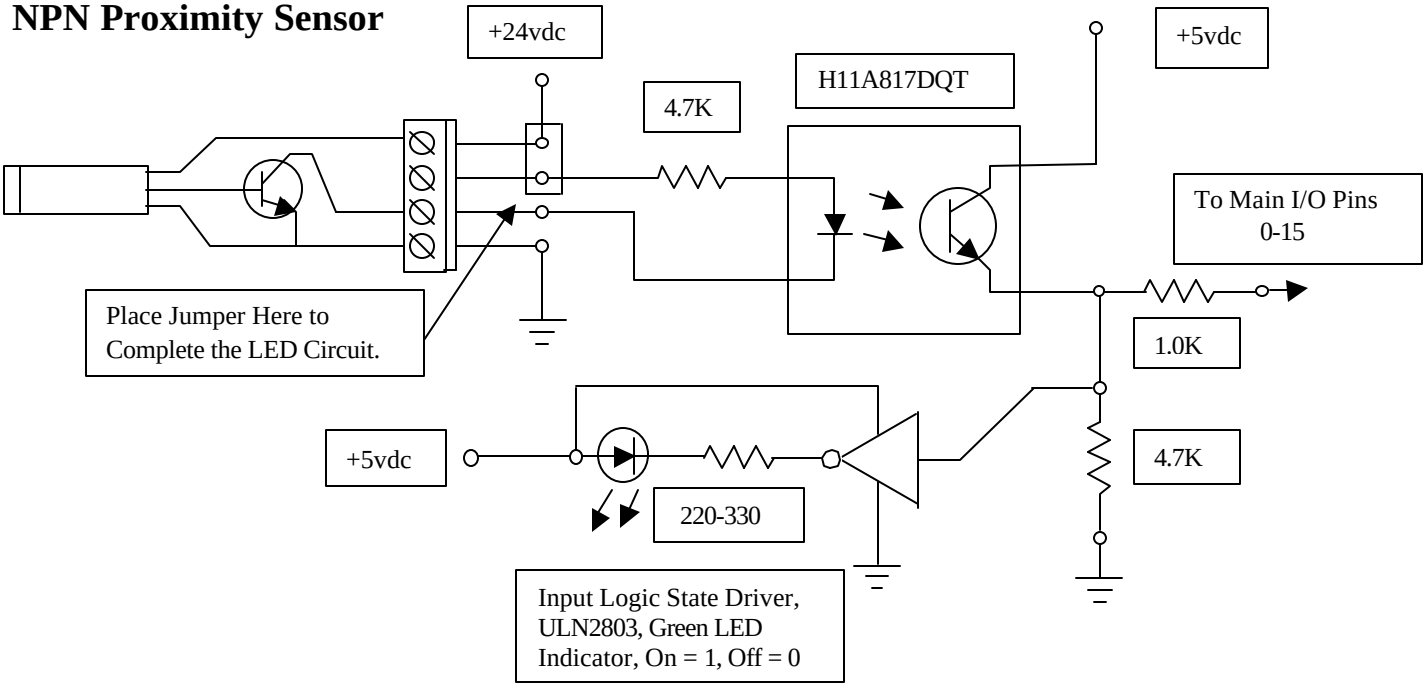
Brown	+24vdc
Black	Signal
Blue	Ground

PNP Proximity Sensor

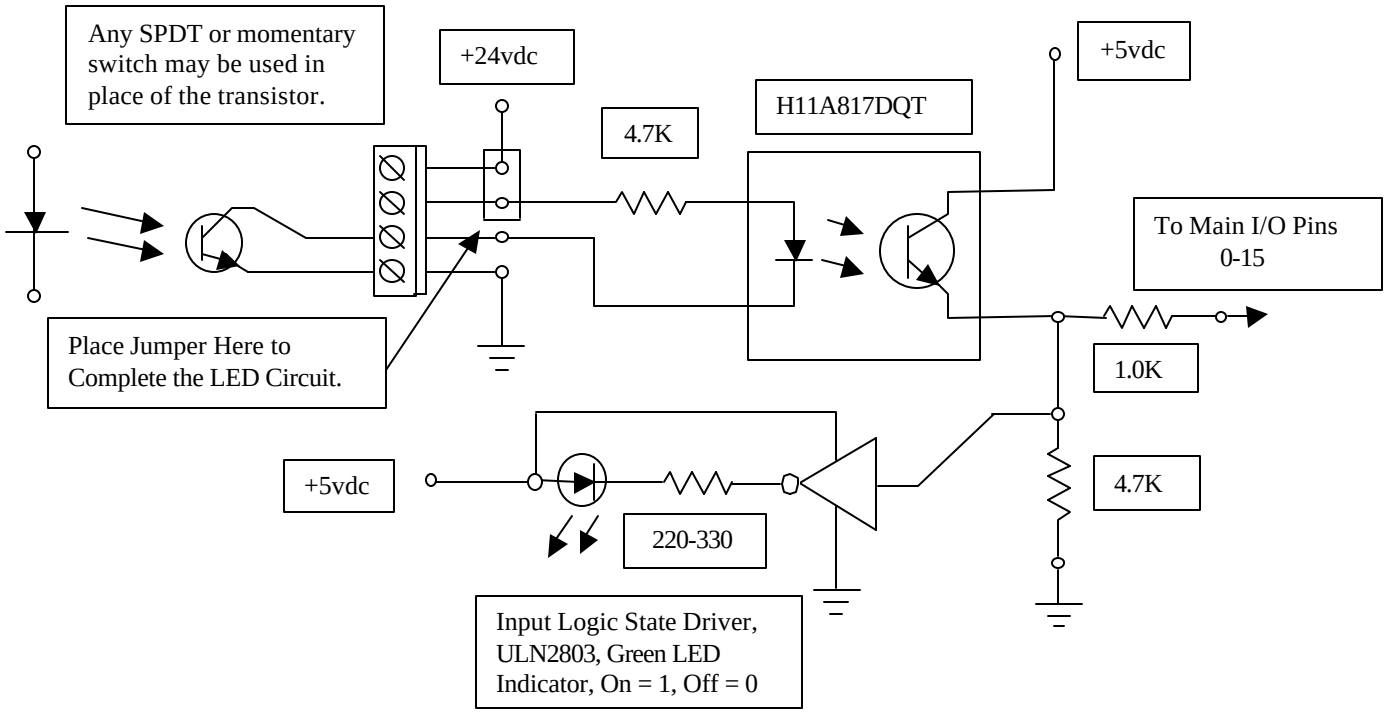
The following schematics are the complete input circuit present on the BSIO16ex module. The input circuit is simply duplicated 16 times, one for each Stamp input pin. The connection labeled "to BSIO pin" is directly connected to the stamp pin corresponding to the I/O address label on the BSIO16ex module. The ground shown is the common buss ground from the main fused power input terminals.



NPN Proximity Sensor

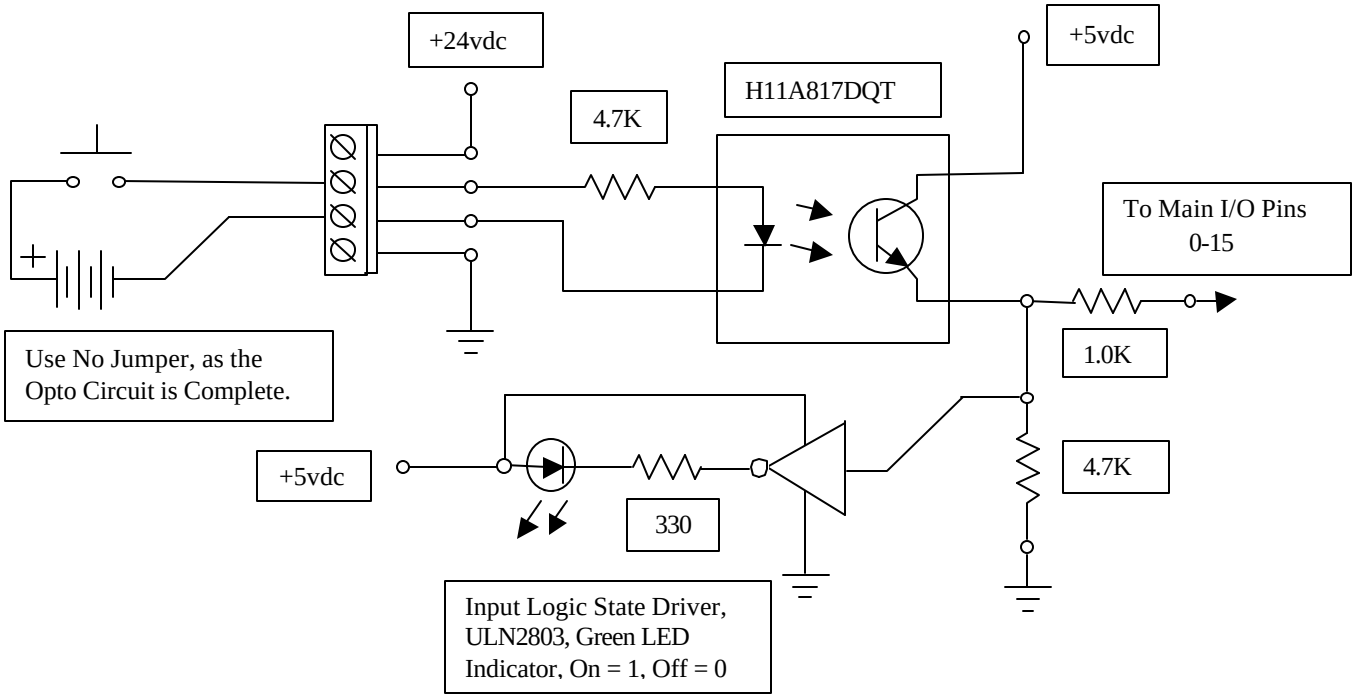


Open Collector Photo-Transistor or Panel Switch



Externally Powered Switch/Input

If the input control does not need power from the BSIO16ex module and is only being used as a logic state select switch, as in the case of using an open collector phototransistor, or a control panel switch, then only two terminals are needed to complete the opto-LED circuit.



The opto-isolator can be used to import a digital signal from another powered device, while maintaining electrical isolation. **Note: Use only DC sources; AC signals will damage the LED in the opto-isolator.** To input an AC signal source, run the AC signal power through a full wave rectifier with an output filter capacitor and maintain correct polarity when connecting to the input terminals. The filtered output should not exceed 24 – 30 volts DC.

Due to the high gain of the Opto-Isolator, a wide range of input voltages will switch the logic state of an input pin. If you use a voltage level lower than 24 volts, then test the input for proper logic operation with this lower voltage.

Pulse Stretching on Inputs

Each input I/O pin or channel is capable of transferring digital logic states, a 1 or a 0. Information may also be transmitted on a pulse width, using the **Pulsin** and **Pulsout** command. *(See the Stamp Programming Manual for Details on this command.)* Due to the slow switching times of some opto-isolators, the width of an incoming pulse will be stretched. The slow switching will add a fixed offset of additional time to an incoming signal.

Example: A pulse width of 100 *us* is sent from one BSIO *(BSIO16 or BSIO16ex)* module using the **Pulsout** command *(using a Pulsout value of 50 in a BSII Stamp)* to a second BSIO16ex module, which is receiving that pulse through the opto-isolator on one of the input terminals. The sent pulse width is read by the second

BSIO16ex module using the **Pulsin** command. This received pulse of 100*us* looks to be 5 *us* longer, showing up as a **Pulsin** count of 52-53 *(plus or minus a few us)*. A **Pulsout** of 20 *(40 us)* will be read as a **Pulsin** of 22-23 *(45 us)* after being sent through the opto-isolator. The amount of pulse stretching is fixed, this offset maybe subtracted in software to retrieve the original value sent.

The datasheet for a H11A817DQT Opto shows a turn-on time of 2.4 *us* and a turn-off time of 2.4 *us*, resulting in a pulse stretch of approximately 5 us.

Input Interrogation via Software

Your program will typically interrogate the input for the logic condition and then branch to some program label, when the tested condition is true. *(See the BSII Memory Map and the Reserved Names* INS, INL, INH, INA, INB, INC, IND, IN0-IN15 *for the I/O Pins in the Stamp Programming Manual.)*

The use of the **If ..Then** command is a typical way to gather information fom an input pin for information. The Basic Stamp has specific reserved names for all input and output pins. For example: "IN0" used below, is a reserved name.

```
'                       All Pbasic Commands are Bold

'_____________________ Sample Section of Your Program  _____________________

'Output Alias Names

DispenseMotor     con 11            'this is a motor control relay on I/O Pin 11


Main_program:
      If IN0 = 0 Then Start_process       'the logic states for your program may be
      If IN0 = 1 Then Stop_process        'different, depending on your situation
      Goto Main


Start_process:
      High  DispenseMotor     'turn motor on
      Goto Main

Stop_process:
      Low DispenseMotor       'turn motor off
      Goto Main
```

The reserved names can be used to make a distinction between inputs and outputs. Inputs may be interrogated as bits, nibbles, byte and word-size numeric values, using these reserved names.

Example: Interrogating the lower byte, pins 0-7 for their decimal value:

```
If INL = 186 Then xyz          'do some function at label xyz
```

Typically the logic conditions within a real program will be complex. Multiple inputs may be tested using **If .. Then** statements to determine what sub routine the program will branch to, to initiate some action, or machine function.

In order to preserve the computing speed of the stamp module, a program should only test for the functions to process next, as a sequence of events. Other repetitive functions, that need to be scanned for constantly, should be written once, yet called for in multiple places by using the **Gosub** command.

Thus the need to look for an operator's input on a Cycle Stop button, may be accomplished under a label of *Cycle_Stop_Button:* which is called from multiple locations within the program. Thus anywhere the program may loop, waiting for something to happen, by inserting a **Gosub** Cycle_Stop_Button in the loop, the operator's pressing of the Cycle Stop button will be caught by the program and then processed at this label.

Note: This example above is not to be confused with an E-stop Function, which should be wired in hardware as a hardwired function. Most E-stop Circuits are normally closed and must be in this state for a machine to run.

The use of **If..Then** logic statements results in hard coded logic sequences, which use very little of the Stamp's RAM variable space. This makes long programs possible with much of the original variable space available for other functions, such as software counters, bit flags, input switch bits set by momentary logic inputs and the implementation of logic sequencers in software.

Most industrial processes occur as a sequence of events, which allows us to write the controlling program in a similar fashion, one that will duplicate the sequence of operation. **If..Then** statements allow us to branch for conditions as they occur, creating different sequences based on conditions within a process.

Distributed Control Architecture

Other configurations in machine control architecture will also facilitate closed loop program operation. Distributed processing is one way of having sufficient speed, processing power and I/O count to control elaborate machines processes or tasks. Take for example: an 8-station turret, with one station empty between the load and unload stations, with 5 station positions left for processing a product in 5 steps.

Each load and unload station with it's own pick and place robot would have its own controller. Each of the 5 processing stations would have a dedicated controller. A master controller would control the turret index rotation and monitor all the other stations. With each rotation or index of the turret, the master controller would send a start process out to all of the stations around the turret.

Each controller would then check a specific station to process the item present *(if an item was present)* and then carry out that process step. When all stations are done with their process step, they would send a "Done" signal to the master controller, which would then index the turret to repeat the process all over again. If the "Done" signal is not received back from all seven processing stations, then the master controller could sound an error to a production operator, reporting which station had a problem on an LCD screen.

Thus with multiple BSIO controllers using a distributed control architecture, one can accomplish a complex task, which is too much for one controller by itself.

Open Control Loops vs. Closed Control Loop I/O Programming

The following program is an example of an open loop program. In an open loop program, the requested actions are not monitored or verified by sensors. Thus one cannot be sure that the action requested actually occurred. To have a more robust process, machine vendors will build control systems with feedback, closing the request loop. For example: sensors may be installed to detect all mechanical positions at the end of their mechanical stroke. In order to build a closed loop system, you may need more inputs.

Sample Program

```
'_____________________________________________________________________________
'
'                       Conveyor Box Sorting Program
'_____________________________________________________________________________

Box_size          var nib           'set variable size
                                    'Output Pin Alias Names
Conveyor_Motor    con 9             'this is a motor control relay
RightLanePush     con 10            'air solenoid
LeftLanePush      con 11            'air solenoid
Dispense_Motor    con 12            'this is a motor control relay
Warning_Light     con 13            'this sinks a light bulb current
Dropgate          con 14            'air solenoid
                                    'Etc… for the rest of the I/O Pins


Start:                          'A fast & slow conveyor will allow space
      Pause 1000                'between boxes as they transfer to next belt
      High  ConveyorMotor       'Turns motor On, This Conveyor is fast
      High  DispenseMotor       'Turns motor On, This Conveyor is slow
      Low   WarningLight        'Turn Warning Light off

Main:
      Gosub Box_detect
      Gosub Box_sort
      Goto Main

Box_detect:
      If  IN4 = 0 Then Stop_Dispense  'Turn off dispense conveyor to stop boxes
      Return

Stop_dispense:
      Low DispenseMotor         'conveyor off, sort current box next
      Box_size = INA            'move sensor array into variable
      Return

Box_sort:                                 'determine box size and then sort
      If Box_size = 0  Then Very_large_box 'all sensors blocked by box
      If Box_size = 1  Then Big_box         'top sensor beam not blocked
      If Box_size = 3  Then Medium_box      'top & middle sensor not blocked
      If Box_size = 7  Then Small_box       'let box go straight through
      If Box_size = 15 Then Start           'no box present, no sensors blocked
      Return
```

```
Very_large_box:
      High  Warning_Light     'alert operators of over size box
      Pause 2000
      If IN5 = 0 then Start   'IN5 is the operators Resume Button Input
      Goto Very_large_box
      Return

BigBox:
      High  RightLanePush     'push big box to right side
      Pause 2000
      Low   RightLanePush     'retract push right air cylinder
      Return

Medium_Box:
      High  LeftLanePush      'push medium box to left side conveyor
      Pause 2000
      Low   LeftLanePush      'retract push left air cylinder
      Return

Small_Box:
      High  Dropgate          'drop box down chute
      Pause 2000
      Low   Dropgate          'close chute
      Return
```

The example machine program above consists of two incoming conveyors in series and two outgoing conveyors, one on each side. Four light beam sensor's shoot diagonally across the box incoming staging area. These four beams are connected to the input pins 0-3 (IN0-IN3), with the least significant bit address being the highest beam above the surface of the conveyor belt to detect the largest of box sizes.

An additional beam (IN4) is shot across the box staging area near the drop gate to detect that a box has arrived. The staging area conveyor never stops, as this lowers the friction when moving a box left or right. Two adjacent conveyors are on the left and right to receive the boxes, pushed left or right. A drop gate allows the small boxes to proceed straight ahead down a chute, when the gate drops. If the box is too big the operator's are notified to remove the box from the staging area and then press a resume button (IN5).

The program is easy to follow and accomplishes the task needing to be automated.

To write the program as a closed loop control mode, additional sensors would be added to the push left-right cylinders on both ends, as well as on the drop gate end positions. Additional **If..Then** statements would also be added to check that the cylinders did arrive at the end stops intended. **For..Next** loops could be used to provide a time out function for checking if a cylinder arrived or not, with the ability to send an error message to the LCD display.

For example: A closed loop version of the Medium _box sorting function would be:

```
LCD con $4054                 'sets serial baud rate for BS2 stamp
                              'you will still have to add the other
                              'LCD commands to your own program

Start:                        'A fast & slow conveyor will allow space
                              'between boxes as they transfer to next belt
      High  ConveyorMotor     'Turns motor On, This Conveyor is fast
```

```
      High  DispenseMotor     'Turns motor On, This Conveyor is slow
      Low   WarningLight      'Turn Warning Light off



                              'start now has the additional code needed to home
Homing Code not shown         'all positions if they are not in the correct,
                              'safe position for starting the process

'**************** Not all of the program code is shown below, only ************
'**************** an expansion of one routine to show the type of  ************
'**************** structure which can implement error messages.    ************

Medium_Box:
      High  LeftLanePush      'push medium box to left side conveyor
      For I = 1 to 20
      If IN6 = 0 Then mb_ext_reverse      'cylinder extension sensor
      Pause 100               'instead of a pause, a gosub to check something
      Next                    'would make better use of the time


mb_extend_error:
      High WarningLight
      serout LCD,15,[ place cylinder failed to extend error message here ]
      If IN15 = 0 Then Start  'IN5 is the Resume Button Input
      Goto mb_extend_error


mb_ext_reverse:
      Low   LeftLanePush      'retract push left air cylinder
      For I = 1 to 20
      If IN7 = 0 Then mb_reset  'cylinder retraction sensor
      Pause 100               'instead of a pause, a gosub to check something
      Next                    'would make better use of the time

mb_return_error:
      High Warning Light
      Serout LCD,15,[ place cylinder failed to retract error message here ]
      If IN15 = 0 Then Start  'IN5 is the operators Resume Button Input
      Goto mb_return_error

mb_reset:
      Return
```

****

In this example of closed loop function programming, the above **For..Next** loop provides adjustable timing delay/pause functions while the **If..Then** statement checks for the action requested to be completed. This replaces the previous open loop control function, which used only a **Pause** statement to allow time for the motion requested, to reach the intended destination before starting the next function.

In this example of closed loop control, if the end of stroke motion is detected prior to the **For..Next** loop expiring the error message is skipped. If the cylinder fails to reach the intended destination then the controller will send an error message to the LCD display and will loop here until corrected. If a box was too heavy, the push cylinder may not move the box to the proper conveyor, creating an error condition.

Alternate Programming Structure for Multiple Tasks

When using a BSIO Module with a large number of I/O the possibility to implement control involving several industrial processes, which may happen asynchronously is possible. Thus several tasks each with there own sequence of events may be programmed to occur independently. Using the correct program structure is the key to making this happen.

Due to the complexity of such control and the coding you may implement, only the outline of the structure will be shown in order to provide this method as a tool for your situation. A machine may have several stations under control with each station executing a different sequential task.

The following items are important in programming such a multiple task controller.

1.) The program must not dwell or pause in any location of the program.
2.) The program will check if a task is ready to take action on a sequence step.
3.) The program uses a variable as sequence counter to keep track of each task.
4.) Each task is given equal CPU access to speed processing.

Example of a Multi-Task Program

Declare a variable for use in each task sequence, you may also have other declarations in your program these are not listed as this a skeleton structure for example purposes

```
tach var byte        'variable to hold drill speed

task_1_seq var nib   'this will allow up to 16 steps for a given task
task_2_seq var nib   'if you need more steps use a byte for the variable size


task_1_seq = 0       'all sequence counters should be set to zero or a number deemed the
task_2_seq = 0       'starting point for controlling the tasks at each station

                     'with several tasks no one part of the program is "the main program"
                     'as all tasks are important

Task_1:
if task_1_seq = 0 then T1Seq_0     'the value of the sequence/task variable will determine where
if task_1_seq = 1 then T1Seq_1     'in the program we wish to jump and thus exclude all other
if task_1_seq = 2 then T1Seq_2     'code, this will speed up the program. The program duplicates
if task_1_seq = 3 then T1Seq_3     'if..then's for the number of sequence steps required for
if task_1_seq = x then T1Seq_x     'the particular process or station you may require more
if task_1_seq =15 then T1Seq_15    'or less than is shown here using a nibble size variable.

T1Seq_0:
if  IN4 = 1 then start_motor
goto Task_2          'your test code will be different, however what is important is
                     'that you are testing for some input condition on which to base some action
                     'if the conditon is not ready for the action desired you move on to task 2

start_motor:         'if the condition you were testing for is correct then you initiate some
hi motor             'action like turning on some motor to perform some function in the process
Task_1_seq = 1       'with a task completed then the task sequence variable is incremented to the
goto Task_2          'next important step, the number may be incremented or maybe change to
                     'another value altogether,  keep in mind that the new value will direct the
```

```
T1Seq_1:            'program to the next code to examine
if IN5 = 1 then insert      'again some code to test for a given condition for some action to be
goto Task_2         'taken if the if..then statement is not true then go on with the program

insert:
hi plug_push        'plug push was assigned a I/O pin which in the end controls a pneumatic
Task_1_seq=2        'cylinder, then increment the task variable sequence number
goto Task_2         'jump to Task 2

'Your …..Code

T1Seq_15:           'at the end of a sequence of events at  given station some event will
if  IN2 = 0 then Seq_reset  'determine the end of the sequence, thus at the end you may need to
goto Task_2                 'reset the mechanics of the machine to a known state to start again


Seq_reset:          'thus previous conditions set may need to be reset depending what
low motor           'happened during the previous steps, this is just an example
low clamp           'the important item is that after all states are reset to begin again
hi gate             'the variable which directs the program sequence must be reset also
Task_1_seq = 0      'with the sequence number for this task back to zero the program
goto Task_2         'will start again

*******************************************************************************

Task_2:

if task_2_seq = 0 then T2Seq_0    'the value of the sequence/task variable will determine where
in
if task_2_seq = 1 then  T2Seq_1   'program we wish to jump for the second task and exclude
if task_2_seq = 2 then T2Seq_2    'all other code, this will speed up the program
if task_2_seq = 3 then T2Seq_3    'the program duplicates if..then test just like Task_1 for the
if task_2_seq =15 then T2Seq_15   'number of sequence step required for this process or station

T2Seq_0:
if IN8 = 1 then drill       'we test for some condition, in this case IN8 detects a clamp is
goto Task_1                 'holding a block of material at this station where task _2 executes
                            'it's control

drill:
hi station drill            'perhaps at this station we will drill a hole in a piece of material
task_2_seq =1               'the drill station is turned on and the sequence number is changed
goto Task_1                 'the item here demonstrated is that the program jumps back and forth
                            'between task 1 and task 2 to check on various conditions which each
T2Seq_1:                    'station needs to accomplish
                            'with the drill on, we want to make sure it is turning
pulsin 10,1,tach            'so we check a tachometer sensor input sensor at P10
If tach < 200 then drill_hole     'we know from previous measurement that a if tach >400 then
drill_fail                        'pulse of less than 200, the drill up to speed, more than 400
goto Task_1                       'indicates a problem

drill_hole:
serout LCD,15,[I,clear]     'if we previously sent a fail message, the drill is to speed now, so
hi hole_cylinder            'clear the LCD screen with drill speed correct, turn on the pneumatic
goto Task_1                 'cylinder which extends the drill into the material

drill_fail:
serout LCD,15,[I,line2, "Drill not to Speed"]   'send a drill motor fail message to the LCD for
goto Task_1                                     'notification to machine operator



T2Seq_2:      'Your test condition code, if true will jump to some action you wish to take
              'perhaps we test to make sure the drill reached the bottom of the hole depth
              'desired before retracting
```

```
goto Task_1

                    'Your Action Code for T2Seq_2 Here:
task_2_seq = 3
goto Task_1

                    'Your Code for T2Seq_3 through T2Seq_14 here

T2Seq_15:           'The code for testing and resulting action to be taken is not shown


task_2_seq = 0      'machine station reset code, to begin process again remember
goto Task_1         'to reset the sequence number variable at the end of the sequence

*****************************************************************************************
```

This type of programming structure is adaptable to multiple tasks. Task_1 calls Task_2 which then calls Task_3 which calls Task_4 which then calls the beginning Task_1 all of which may have a different number of tasks to perform at their own station. Thus these may operate asynchronously within the program.

If your program is large, the call for another task maybe a **run** 1, **run** 2 or **run** 3 rather than a **goto** command. Run refers to the program number in which each task is located. If this is the case remember to save or write the task_X_seq variable to **scratch pad ram** and then retrieve it upon entering each program. This will allow you keep track of all sequence numbers for each task. *(See the Get, Put and Run Commands in your Stamp Programming Manual.)*

Multi-Task Overview

The overall structure for multi-task coding maybe viewed as a series of gosub's which call the specific task sequence via the task number variable. The coding below is only an example of the programming architecture. It is impossible to cover all the possible variations that maybe encountered in the industrial environment.

```
T0 var byte   'T0 is the variable which holds the sequence number for directing the program
T1 var byte   'T1 is the variable for Task_1 just as T0 is for Task_0, each task has
T2 var byte   'its own variable, continue till all tasks have there own variable
T3 var byte   'use only the size needed for each Task sequence,
TN var byte   'use a bit for a two state variable, nib for up to 16 steps, byte for 256.

T0 = 0        'Initialize all variables to the correct value for starting the machine
T1 = 0
TN = 0

Machine_homed var bit       'A machine many times is not allowed to start if the machine
                            'is not in a known state. Once the machine is homed
Machine_homed = 0    'the bit can be changed to a logic state of 1.
                     'Thus one should add a subroutine to home the machine or process.
                     'All of this depends on the design of your machine/process.
                     'Once any task is initiated the home bit should
                     'be set to 0, for example.

gosub  home_machine


Loop:

       gosub  Task_0
```

```
       gosub  Task_1
       gosub  Task_2
       gosub  Task_3
       gosub  Task_N        'The number of task's are dependent on your application.

       gosub  Task_0_reset  'Each task sequence may have its own reset sequence
       gosub  Task_1_reset 'based on the value of the task sequence variable
       gosub  Task_2_reset  'to enter a reset, you may assign an operator button
       gosub  Task_N_reset 'to allow the reset of the a industrial process.

       gosub  Gobal_reset   'Or one may have a global machine reset which
                            'will reset all processes, with the push of a button
                            'assigned to an input.
       goto Loop

Task_0:
       branch T0, [T0_a,T0_b,T0_c,T0_d,T0_x]    'The value of T0 will jump the program
                                                'to the correct sequence, skipping steps
                                                'already accomplished.

       T0_a:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T0=1          'Remember that the task variable must be set to the next
                            'step for branch to work
              Return        'Return to multi-task loop

       T0_b:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T0=2          'Set variable to next branch location
              Return        'Return to multi-task loop

       T0_c:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T0=3          'Set variable to next branch location
              Return        'Return to multi-task loop

       T0_x:                'Continue adding steps till all sequence steps are covered
              Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T0=0          'Reset task sequence variable to zero
              Return        'Return to multi-task loop

Task_1:
       branch T1, [ T1_a,T1_b,T1_c,T1_d,T1_x]  'The value of T1 will jump the program
                                                'to the correct sequence, skipping steps
                                                'already accomplished.

       T1_a:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T1=1          'Remember that the task variable must be set to the
                            'next step for branch to work
              Return        'Return to multi-task loop

       T1_b:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T1=2          'Set variable to next branch location
              Return        'Return to multi-task loop

       T1_c:  Test_Code     'If test code is true continue on to action code,
```

```
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T1=3          'Set variable to next branch location
              Return        'Return to multi-task loop

       T1_x:                'Continue adding steps till all sequence steps are covered
              Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T1=0          'Reset task sequence variable to zero
              Return        'Return to multi-task loop


Task_2:
       branch T2, [ T2_a,T2_b,T2_c,T2_d,T2_x]  'The value of T2 will jump the program
                                                'to the correct sequence, skipping steps
                                                'already accomplished.

       T2_a:
              Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T2=1          'Remember the task variable must be set to the next step
                            'for branch to work
              Return        'Return to multi-task loop

       T2_b:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T2=2          'Set variable to next branch location
              Return        'Return to multi-task loop

       T2_c:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T2=3          'Set variable to next branch location
              Return        'Return to multi-task loop

       T2_x:                'Continue adding steps till all sequence steps are covered
              Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T2=0          'Reset task sequence variable to zero
              Return        'Return to multi-task loop


Task_3:
       branch T3, [ T3_a,T3_b,T3_c,T3_d,T3_x]  'The value of T1 will jump the program
                                                'to the correct sequence, skipping steps
                                                'already accomplished.

       T3_a:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T3=1          'Remember that the task variable must be set to the next
                            'step for branch to work
              Return        'Return to multi-task loop

       T3_b:  Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T3=2          'Set variable to next branch location
              Return        'Return to multi-task loop

       T3_c:  Test_Code     'If test code is true continue on to action code,
```

```
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T3=3          'Set variable to next branch location
              Return        'Return to multi-task loop

       T3_x:                'Continue adding steps till all sequence steps are covered
              Test_Code     'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              T3=0          'Reset task sequence variable to zero
              Return        'Return to multi-task loop

Task_N:
       branch TN, [ TN_a,TN_b,TN_c,TN_d,TN_x]  'The value of TN will jump the program
                                                'to the correct sequence, skipping steps
                                                'already accomplished.

       TN_a:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              TN=1          'Remember that the task variable must be set to the next
                            'step for branch to work
              Return        'Return to multi-task loop

       TN_b:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              TN=2          'Set variable to next branch location
              Return        'Return to multi-task loop

       TN_c:  Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              TN=3          'Set variable to next branch location
              Return        'Return to multi-task loop

       TN_x:                'Continue adding steps till all sequence steps are covered
              Test_Code    'If test code is true continue on to action code,
              Return        'If test code shows false then return to multitask loop
Action Label: Action_Code
              TN=0          'Reset task sequence variable to zero
              Return        'Return to multi-task loop
```

Your application may not have multiple tasks, however the structure used here is also applicable for single task control solutions. The use of a sequence variable aids program direction and reset sequences.

LCD Port

Many times during programming, the process of writing code is made easier by exporting the task variables to an LCD screen, thus one can see what the current state of the process or machine is in. One line of code may be inserted at the beginning of the main loop, to send this data to the LCD, as all tasks are only tested/executed once on each loop, if the process is ready for the execution.

An LCD port is provided in the center of the BSIO16ex module. A mating connector is provided with the Module to facilitate the use of LCD's wired to a serial backpack module. The typical serial backpack used for LCD's is one sold by Scott Edwards Electronics. The connector is pin for pin identical to the input connector on the serial backpack.

To provide some selection, this port has a jumper address option. Several auxiliary I/O pins are available to be used on the module. The LCD port will prevent the use of output pins 12, 13, 14, or 15 for other control purposes. **NOTE: Use one jumper only, as two will cause I/O pin contention and excessive current draw, damaging the BS2p40.**

Aux. Port and I2C Port

Two additional direct I/O ports are available on this board. These electrical connections are direct to the I/O pins on both the Main and Auxiliary pins. The Logic indicator LED's for Aux I/O port will show up on the Red LED's when these pins are driven either by the Stamp, or an external driver if they are used as inputs.

The I2C Port Lines are direct to the Stamp Main I/O Pins. However to use this I2C port the jumpers for the specific pair of I/O Pins must be moved from the headers to the left of the Stamp module to the identical labels on the Header above the I2C Port Connector. This will disable the I/O lines connected to the Opto-Isolators and will prevent there use for anything other than I2C communications with I2C devices off the BSIO module. These lines are not exclusively I2C, rather other forms of communications are also possible, SPI microwire, (3 wire) or one wire devices may also use this port. *(See the Stamp Programming Manual 2.0 for more details on I2C commands.)*

Note: Upon moving the RED jumpers, these I/O lines will now change flavor from a 10K pull-down logic to a 4.7K pull-up logic to facilitate I2C communications. No green LED Logic indicators will be available for the I2C lines while being used for this purpose. Power is also available on these port connectors to power up external 5 volt TTL, CMOS electronics. *(See the printed circuit board Labels next to these connectors for the specific I/O pins available.)*

Special Function circuits may be added to these ports for additional control. ADC's, DAC's, PWM modules, external RAM or EEPROM, solid state temperature sensors, analog pressure sensors and thermocouple inputs are just some of the extra features that may be added via these port connections.

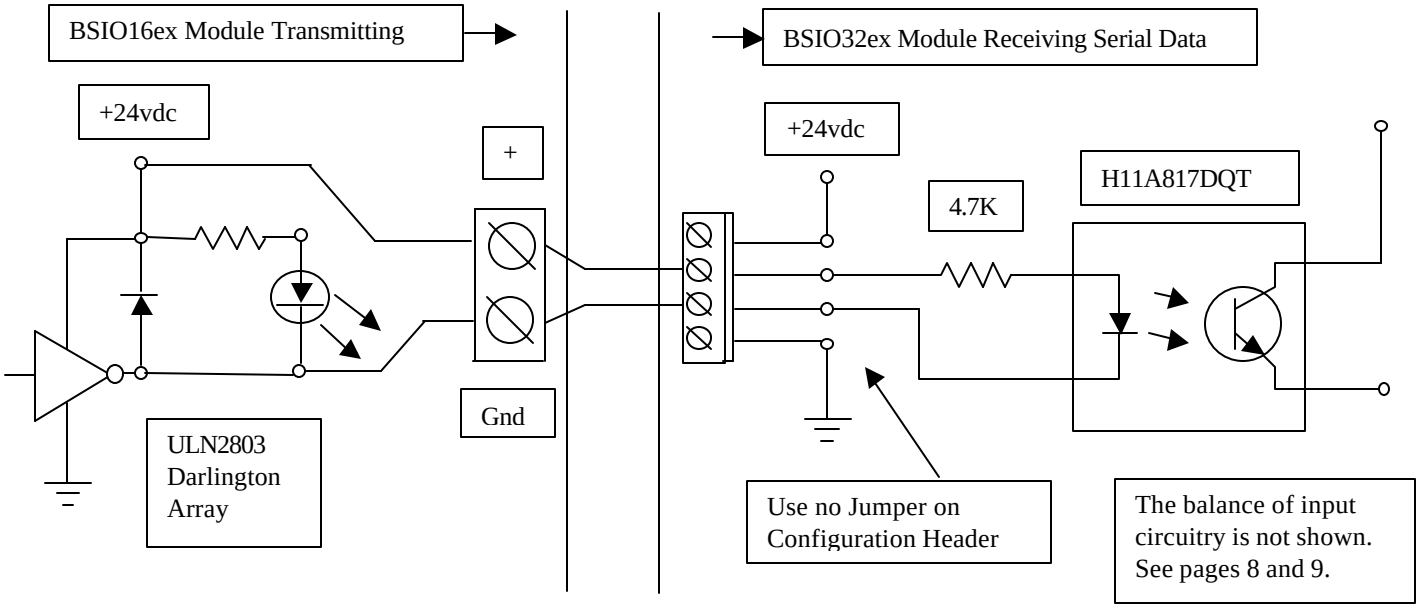
Direct I/O Serial Communication

All of the direct I/O pins used in either the LCD port or the Expansion Ports, may be used for bi-directional serial communication with off board electronics.

Note: The use of an in-line current limiting resistor 1 – 2 K should be installed on the serial line to prevent current overload of a direct I/O pin should the serial line be driven at both ends to opposite logic states. Failure to install a current limiting resistor could result in damage to both the Stamp module and the external electronic circuitry.

Serial Communications Via Opto-Isolated I/O

The Opto-Isolation inputs and the open collector outputs will maintain the polarity of logic states from one module to another. Thus the Basic Stamp on one BSIO16ex module may transmit inverted serial data via the standard open collector output through the opto-isolator to the Basic Stamp in another BSIO16ex or BSIO32ex module.



A logical "1" placed on the output driver of the transmitting module will translate to a logical "1" on the input pin of the receiving module. A BSIO16 module may also transmit serial data to a BSIO16ex module using this wiring configuration. **Note: This mode of serial communication is not bi-directional.**

DIN Rail Mounting

The plastic mounting feet under the module fit a standard DIN rail. This rail material maybe purchased from Newark Electronics and other electronic parts vendors.

Newark Electronics 1-800-463-9275

Stock# 96F6501 Aluminum DIN Rail (1000mm)
Stock# 92N4515 Aluminum DIN Rail 3 feet

Your automation projects may need other components with DIN rail mounting capability. All sorts of components come with DIN rail mounting, Power Supplies, Sensor Amplifiers, SSR drivers, relays and others. Newark offers many other products for Industrial Control and Automation. A web search will reveal other companies offering similar products.

Keyence, Honeywell, Balluff, Banner, Omron, IDEC, NaiS, CRYDOM, GE and Allen Bradley all offer components, like optical sensors, proximity sensors, control panel switches, solid state relays and magnetic

relays for the industrial environment. SMC and Bimba have pneumatic cylinders, flow controls and solenoids for use in mechanical motion generation and control using compressed air.

Basic Stamp Components

For other required components, like Programming Cables, Stamp Programming Manual and BASIC Stamp Modules, these may be ordered from Parallax, Inc.

Contact Information

Digital Design Solutions, Inc.
128 Currituck Ct.
Morresville, N.C. 28117
Phone: (704) 500-6024
For Applications Engineering
Please E-mail Ron Anderson at: ronaldsa@earthlink.net

Parallax, Inc.
599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Toll-Free Sales: (888) 512-1024
Office/Technical Support: (916) 624-8333
Fax: (916) 624-8003

Web Sites:
Parallax, Inc. http://www.parallaxinc.com

Notes: