

# AVR Firmware: GPIO, Version 5 (Beta)

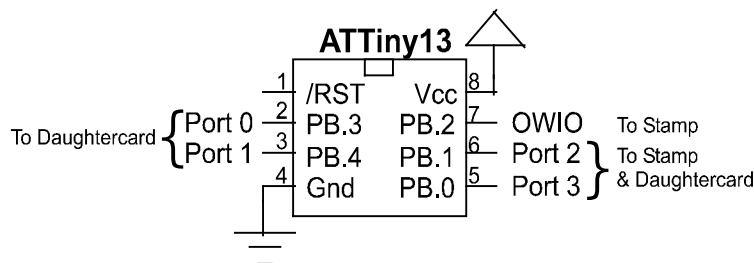
## Introduction

This document describes the use of AVR firmware that is used in conjunction with the BS2p/BS2pe BASIC Stamp motherboard. This firmware can be uploaded to either or both of the motherboard's two AVR coprocessors as file **GPIO5B.hex**. Once loaded, the coprocessor is capable of communicating with the Stamp using PBASIC's **OWOUT** and **OWIN** commands. This communication takes place on the AVR's OWIO pin (see illustration below) to read data from, and write data to, the other four I/O pins. These four pins, two of which are shared with the Stamp, also connect to an attached daughterboard. By utilizing the capabilities of the AVR coprocessor, interaction is afforded with the daughterboard in ways that augment the Stamp's capabilities and offload from it the more mundane and processor-intensive tasks.

The functions available with this firmware include:

- Digital input on all four ports, with optional internal pullups.
- Digital output on all four ports.
- 1 MHz frequency counter on all four ports, with optional internal pullups.
- Up to 37.5 KHz PWM output on two ports.
- Modulated outputs under BASIC Stamp control, including PWM for IR remote applications.
- 10-bit analog-to-digital input on two ports.
- Analog comparator input on three ports, comparing with either the fourth port or a 1.1- volt reference.
- 32 bytes of RAM, which can be written and read.

In addition, the type and frequency of the PWM outputs can be configured on the fly.  
The AVR (Atmel ATTiny13) pinout is shown below:



OWIO connects to the Stamp and has a pull-up resistor to Vdd. Communication is bi-directional via a protocol using open-collector signaling. Ports 2 and 3 also connect to the Stamp without external pull-ups, as well as to an attached daughtercard. Ports 0 and 1 connect to an attached daughtercard only. Ports 0 and 1 are capable of analog input. Ports 2 and 3 are capable of unattended PWM output. Ports 0, 1, and 2 can be used as positive comparator inputs, comparing with either Port 3 or an internal 1.1-volt bandgap reference. All ports can be used for digital I/O and frequency counting.

## Command Protocol

Communication with the AVR is via one-byte commands sent using the Stamp's **OWOUT** statement, possibly followed by additional data bytes or by reads using **OWIN**. An example command might be:

```
OWOUT Owio, 0, [$25]
```

This statement will write a digital "1" to port 2, causing it to go high. Some commands are used to read from the AVR. An example would be:

```
B VAR Bit
OWOUT Owio, 0, [$10]
OWIN Owio, 4, [B]
```

This reads a one-bit value (high or low) from Port 1 into variable B.

Any reset (low pulse lasting longer than 160 microseconds) sent via the OWIO pin will reset the AVR's protocol state machine, interrupting any transaction in progress (with one exception, described under "Counter Input" below). It will not affect the states of any of the pins, however. A reset can be incorporated into an **OWOUT** statement by choosing the second argument appropriately. For example:

```
OWOUT Owio, 1, [$14]
```

resets the protocol engine before writing a zero to Port 1. A reset is usually a good idea in the first transaction of a Stamp program. It should also be considered when communication with the AVR is infrequent or done in electrically noisy environments. A too-liberal use of resets, however, can not only slow a program down, but it can also mask program errors associated with AVR communications.

Finally, the AVR comes out of a hardware reset more slowly than the Stamp does. So don't start talking to it right away in your Stamp program. To make sure the AVR is ready for communicating, put a 5ms PAUSE at the beginning of your Stamp program:

```
PAUSE 5
```

This will prevent out-of-the gate misfires.

## Firmware Identification

To identify the firmware currently extant in the AVR (assuming it uses the same protocol), send the following command (\$DD):

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

The AVR will respond with three bytes of data. The first two are characters representing the name of the firmware. The third is a version number. This information can be used by a Stamp program to make sure the correct AVR firmware is loaded. The following program snippet, where **I**, **J** and **K** are byte variables, prints out this information:

```
OWOUT Owio, 0, [$DD]
OWIN Owio, 0, [I, J, K]
DEBUG " Device: ", I, J, ", Version: ", HEX2 K
```

For this beta firmware, the following line is printed:

```
Device: GP, Version: 5B
```

## Outputs

Outputting signals from the AVR is a one-step process. It involves sending the AVR a command telling it what's to be done. Some commands require an additional data byte after the command byte. If either or both of Ports 2 and 3 are used as outputs, their corresponding shared Stamp pins must be set to inputs to avoid bus conflicts.

By setting the "modulate" flag (**m** in the commands below) to a **1**, the AVR's output driver for the chosen pin can be modulated by the **Owio** pin after the command has been issued. See **Modulated Output** below.

### Digital Output

The command for causing a port to drive high or low is as follows:

0	m	Addr	0	1	0	N
---	---	------	---	---	---	---

**Addr** is the two-bit address (00 = 0, 01 = 1, 10 = 2, 11 = 3) of the destination port. **N** is the bit value (0 or 1) to write. A zero drives the pin low; a one drives it high.

The following program segment, sets all four pins low:

```
FOR I = 0 TO 3
  OWOUT Owio, 0, [I << 4 + $04]
NEXT
```

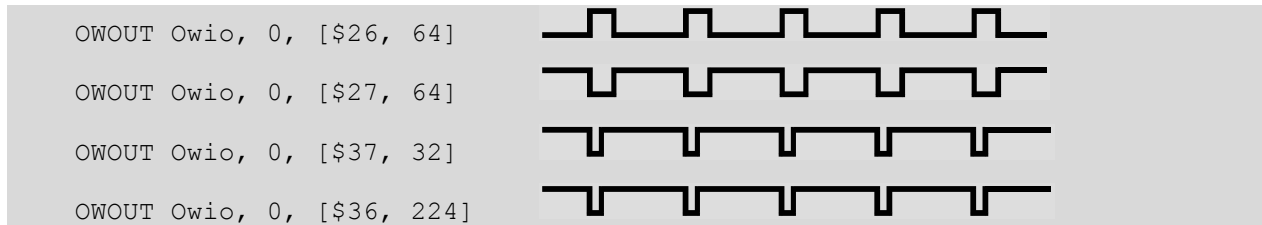
### PWM Output

Ports 2 and 3 can be configured as pulse-width modulated outputs having 256 possible duty cycles from 0% to 99.6% (or 100%: see below). A PWM output can be low-pass filtered and used to form a rudimentary digital-to-analog converter (DAC). Or, it can be used with an appropriate driver to modulate inductive loads directly, without filtering. The maximum PWM frequency attainable with the AVR is 37.5 KHz, which can be filtered with fairly small-valued components. The PWM output command has the following format:

0	m	Addr	0	1	1	Inv
---	---	------	---	---	---	-----

<b>Data byte: \$00 - \$FF (0 – 255)</b>
---

**Addr** is the two-bit address (10 = 2, 11 = 3) of the destination port. Writes to Ports 0 and 1 are ignored, since they do not support PWM output. The **Inv** bit selects the sense of the PWM: A zero here selects positive-going pulses; a one selects negative-going pulses. The data byte selects the desired relative pulse width. Example commands and the pulse trains they produce are shown below:



The first two lines write to Port 2; the second two, to Port 3. Lines 1 and 4 create positive pulses; lines 2 and 3, negative pulses. Notice that a positive pulse with a duty cycle of  $224/256 = 87.5\%$  looks a lot like a negative pulse with a duty cycle of  $32/256 = 12.5\%$ .

It is also possible to select from several fixed overall frequencies, using the frequency command:

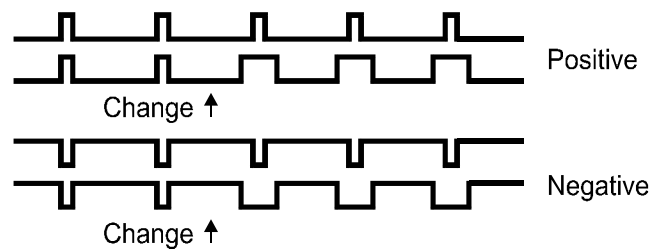
<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>Freq</b>
----------	----------	----------	----------	-------------

**Freq** is a four-bit number ranging from 0 through 9 that specifies the actual PWM frequency. The selected frequency applies to both Ports 2 and 3 equally. It is not possible to specify a different frequency for each port. The **Freq** values and their associated nominal frequencies are:

<b>Freq</b>	<b>Frequency</b>	<b>PBASIC Code</b>
0	37,500.00 Hz	OWOUT Owio, 0, [\$F0]
1	18,823.53 Hz	OWOUT Owio, 0, [\$F1]
2	4,687.50 Hz	OWOUT Owio, 0, [\$F2]
3	2352.94 Hz	OWOUT Owio, 0, [\$F3]
4	585.94 Hz	OWOUT Owio, 0, [\$F4]
5	294.12 Hz	OWOUT Owio, 0, [\$F5]
6	146.48 Hz	OWOUT Owio, 0, [\$F6]
7	73.52 Hz	OWOUT Owio, 0, [\$F7]
8	36.62 Hz	OWOUT Owio, 0, [\$F8]
9	18.38 Hz	OWOUT Owio, 0, [\$F9]

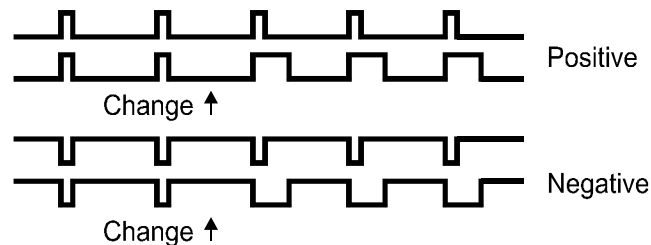
Remember, these are nominal frequency values. They are derived from the AVR's internal RC clock, so they can vary as much as  $\pm 10\%$ . Frequency values larger than 9 are ignored, and no changes are made if a larger value is attempted.

Finally, the odd-numbered frequency values cause the PWM to behave somewhat differently than the even-numbered ones do. The odd-numbered values designate “phase correct” pulses. This is to say that when the duty cycle changes, the change is made symmetrically about the center of the pulse, like this:



This type of PWM is often used for DC motor speed controllers or other inductive load drivers, where phase shifts may be undesirable.

In the even-numbered frequencies, the leading edges of the pulses line up, regardless of any changes made to the duty cycle, thus:



This type of PWM is completely adequate for low-pass filtering to obtain a voltage output.

The other difference between the two PWM modes is the base length of each pulse. In the even-numbered modes, the base length is 256 units. So duty cycles can range from 0/256 to 255/256, positive or negative. The odd numbered modes have a base length of 255 units. So duty cycles can range from 0/255 to 255/255, positive or negative.

## Modulated Output

When the **m** bit in the **Digital Output** and **PWM Output** commands is set to a **1**, the command executes as specified, except that the AVR pin is left in the input state. Thereafter, whenever the **Owio** pin is brought low, the chosen output pin is driven to its specified state (i.e. high, low, or PWM). When the **Owio** pin is brought high, the chosen output pin floats again. This continues until a negative-going pulse shorter than 3μs is detected on **Owio**, at which point the chosen pin remains floating, and the AVR resumes accepting commands on **Owio**.

**Note:** While the AVR is modulating the chosen output pin, it is insensitive to long reset pulses on **Owio**. The *only* way to terminate modulation is to send a short pulse, less than 3μs, as described above. Because of this protocol, the modulation of the chosen output pin will experience a 4μs delay on both rising and falling edges from **Owio**. Therefore, it is not possible to modulate at a rate faster than 125KHz.

Modulated output has two main applications:

1. Transferring pulses on **Owio** to any of the AVR's four I/O pins, rather like a digital multiplexer. This makes it possible, for example, to send serial data to pins **A0**, **A1**, **B0**, or **B1**, which do not

have connections in common with the BASIC Stamp chip. It also enables all the AVR pins (which have no series protection resistors) to drive loads that the more protected Stamp pins cannot, such as the input to the Parallax Servo Controller.

2. Modulating IR LEDs for communication with an infrared detector. Since the nominal 37.5KHz PWM output of the AVR is close to the 38KHz detection frequency of these detectors, it is possible to send serial data to them by modulating a 50% duty cycle PWM output with the **SEROUT** command. this makes wireless communication between two motherboards, or from a motherboard to a remote-controlled device easy.

Following is an example program that communicates with a Parallax Servo Controller to read its version number. It demonstrates not only the ability to modulate an AVR output pin, but also the fact that a shared pin (i.e. **A2**, **A3**, **B2**, or **B3**) can also be used as an input pin to the Stamp while the AVR is in the modulated output state:

```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

Owio  PIN 10          ' AVR's (socket A) GPIO5 command pin.
Sinp  PIN 11          ' Serial input pin (same as A2).

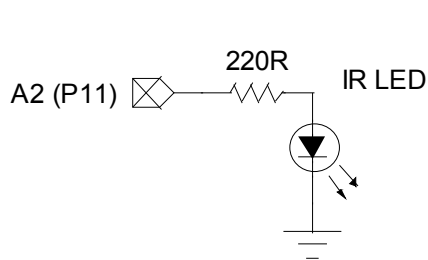
Baud  CON 396         ' Constant for 2400 baud.

buff  VAR Byte(3)     ' Temporary variable.

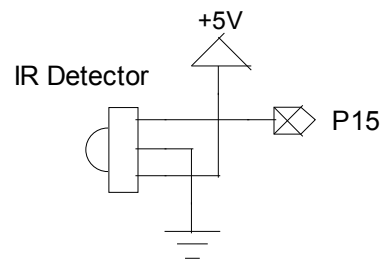
PAUSE 10              ' Wait for AVR to reset
OWOUT Owio, 1, [$64]  ' Set A2 (P11) to a modulated "low" output.

FindPSC:              ' Get the version number of the PSC.
  DEBUG "Finding PSC", CR
  SEROUT Owio, Baud+$8000, ["!SCVER?", CR]  ' Send the PSC "Version"
                                           '   command via Owio.
  SERIN Sinp, Baud, 500, FindPSC, [STR buff\3]  ' Get the response
directly                                           '
                                           '   from Sinp.
  DEBUG "PSC ver: ", buff(0), buff(1), buff(2), CR  ' Display the result.
  HIGH Owio                                           ' Send a short pulse to
                                           '   Owio to terminate
                                           '   modulated output.
  PULSOUT Owio, 1                                     ' Restore Owio to an
  INPUT Owio                                           '
input.
  STOP
```

The next example shows how to drive an infrared LED with a carrier frequency of 37.5Khz, modulated by serial data for wireless communication with another BASIC Stamp. For this example, we need one MoBoStamp-pe and another BASIC Stamp board, wired as shown below:



MoBoStamp-pe



Second BASIC Stamp

Here's the code that drives the LED:

```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

Owio    PIN 10                'Use 10 for socket "A"; 6 for socket "B".

PAUSE 10
OWOUT Owio, 1, [$66, $80]     'Set up A2 for modulated, 50% PWM output.
DO
  SEROUT Owio, 396 + $8000, ["Test: ", DEC B0, CR] 'Output to LED.
  B0 = B0 + 1
  PAUSE 100
LOOP
```

This is the code used by the receiving BASIC Stamp to capture and display the IR serial data:

```
' {$STAMP BS2}
' {$PBASIC 2.5}

DO
  SERIN 15, 396, [WAIT("Test: "), DEC W0] 'Receive serial data from detector.
  DEBUG DEC W0, CR                        'Display the numerical portion.
LOOP
```

For serial I/O using modulated IR, be sure to use a baud rate of 2400 or less. This will give the IR detector about 15 cycles of the 37.5KHz carrier per bit. Higher baud rates would leave fewer carrier cycles per bit and could affect reception reliability.

## Inputs

Reading signals from the AVR's pins is a two-step process. First the appropriate command is sent via **OWOUT**. Then **OWIN** is used to get the actual data. If you are reading signals from a daughterboard on Ports 2 and/or 3, be the corresponding shared Stamp pin(s) are configured as input(s) to avoid bus conflicts.

### Digital Input

The following command format is used for reading digital pin values:

0	u	Addr	0	0	0	0
---	---	------	---	---	---	---

**Addr** is the two-bit address (00 = 0, 01 = 1, 10 = 2, 11 = 3) of the source port. **u** is a flag indicating whether to enable the internal pullup on this pin. If set to a **1**, the pullup is enabled; if reset to a **0**, it is disabled. Once issued, **OWIN** must be used in bit mode to read the state of the chosen port pin.

The following program segment reads the state of pin 3:

```
State VAR Bit
OWOUT Owio, 0, [$30]
OWIN Owio, 4, [State]
```

In some cases it may be desired just to tri-state the port pin or to enable the pullup without reading it. This can be accomplished by sending a reset after the command byte, as follows:

```
OWOUT Owio, 2, [$20]
```

All this does is to make Port 2 float as an input pin. **\$60** would have done the same, while enabling the pin's pullup.

## Counter Input

By invoking the counter input command, pulses arriving on any selected AVR port pin may be counted over a programmed time interval. The AVR can count pulses arriving at up to a 1MHz rate. Minimum high and low times for these pulses are 500ns each. The counter input command has the following format:

<b>0</b>	<b>u</b>	<b>Addr</b>	<b>0</b>	<b>0</b>	<b>H</b>	<b>L</b>
----------	----------	-------------	----------	----------	----------	----------

**Addr** is the two-bit address (00 = 0, 01 = 1, 10 = 2, 11 = 3) of the source port. **u** is a flag indicating whether to enable the internal pullup on this pin. If set to a **1**, the pullup is enabled; if reset to a **0**, it is disabled. **H** and **L** determine which bytes (high-order and/or low-order) of the final count to return. At least one of these bits must be a one. If only the low-order byte is selected and if the actual count is greater than 255, 255 is returned. If only the high byte is selected, and the actual count is greater than 65535, 255 is returned. If both bytes are selected, and the actual count is greater than 65535, 65535 is returned.

This command starts the counting process immediately. Counting continues until the next falling edge from the Stamp on **OWIO**. This will typically be a reset pulse. But unlike all other resets, this one does not reset the protocol engine: it merely stops counting and sets up to send the count byte(s) back to the Stamp via its **OWIN** command. It's most convenient just to use an **OWIN** prefaced by a reset pulse (*i.e.* **OWIN Owio, 1, ...**).

The following example counts pulses on Port 2 for a duration of 50ms and computes the actual frequency, which then gets printed out:

```
Result VAR Word
OWOUT Owio, 0, [$23]
PAUSE 50
OWIN Owio, 1, [Result.HIGHBYTE, Result.LOWBYTE]
DEBUG "Frequency: ", DEC Result / 50
DEBUG ".", DEC2 Result // 50 * 2, " KHz", CR
```

Remember that the actual number of bytes read by **OWIN** *must* agree with the number requested by the **H** and **L** parameters.

## Comparator Input

An analog voltage on any of Ports 0, 1 or 2, may be compared with either an internal 1.1-volt reference, or with an analog voltage on Port 3. The following command performs the comparison:

0	0	Addr	1	Ref	0	0
---	---	------	---	-----	---	---

**Addr** is the two-bit address (00 = 0, 01 = 1, 10 = 2) of the input port. (If Port 3 is chosen, the result of the comparison will always be zero.) **Ref** determines what to compare the pin to. A zero selects Port 3; a one, the internal 1.1-volt reference.

After this command is issued, a 1-bit **OWIN** should be used to read the result of the comparison. A "one" means the **Addressed** port pin was higher than the reference; a "zero", equal or lower. In the following example, Port 1 is compared with Port 3. The result is read into the Bit variable **C**:

```
C VAR Bit
OWOUT Owio, 0, [$18]
OWIN Owio, 4, [C]
```

## Analog Input

An analog voltage on either Port 0 or Port 1 may be converted to a digital value by the AVR's 10-bit A-to-D converter and read by the Stamp. The command for doing so is as follows:

0	0	Addr	1	Ref	H	L
---	---	------	---	-----	---	---

**Addr** is the two-bit address (00 = 0, 01 = 1) of the analog input port. **Ref** determines the voltage reference source (highest voltage) for the A-to-D conversion. A zero selects Vdd; a one, the internal 1.1-volt reference. **H** and **L** determine how to return the digitized value. At least one of these bits must be a one. The following table defines how **H** and **L** are interpreted:

H	L	Actual 10-bit value (ADC)	Returned Bytes(s)
0	1	0 – 255	ADC[7..0]
		256 - 1023	255
1	0	0 - 1023	ADC[9..2] ( <i>i.e.</i> ADC / 4)
1	1	0 - 1023	ADC[9..8], ADC[7..0]

As soon as the command is issued, the desired A-to-D conversion begins. Because this conversion may not be finished before the next read, it is necessary to poll the AVR by reading a single bit before the conversion result can be read. If this bit is a "one", the AVR is still busy. If it's a "zero", the conversion is complete, and the result byte(s) may be read.

The following example reads the voltage on Port 0, using Vdd as a reference, and returns the entire 10-bit value:

```
Voltage VAR Word
Busy VAR Bit
OWOUT Owio, 0, [%00001011]
DO : OWIN Owio, 4, [Busy] : LOOP WHILE Busy
OWIN Owio, 0, [Voltage.HIGHBYTE, Voltage.LOWBYTE]
```

If you use Vdd as the voltage reference, be sure you have a reliable voltage source. If the motherboard is jumpered to use the USB's 5-volt source as Vdd, this voltage could be as low as 4.2 volts. Under these circumstances, it would be better either to use 3.3V for Vdd or to lower the analog input voltage with a resistor divider and use the internal 1.1-volt reference. An alternative would be to add a 2.5V bandgap reference to the other analog input and read both the unknown voltage and the 2.5-volt reference voltage, as compared to Vdd. Then you can compute the unknown voltage in such a way that the Vdd terms cancel.

**Important Note:** No voltage input on an AVR analog input pin may exceed Vdd by more than 0.5 volts, regardless of which reference you use. Use a resistor divider, if necessary, to keep such voltages in range.

## RAM Writes and Reads

The AVR has 32 spare bytes of RAM available. The values stored here are zeroed on a hardware reset, but persist across multiple protocol resets. This section describes how to access the RAM from PBASIC.

### RAM Writes

To write a single byte to RAM, use the **Enter** command (\$En):

1	1	1	0	0	Addr
---	---	---	---	---	------

The number **Addr** can range from **0** to **7** and represents a shorthand notation for writing a *single byte* to the address represented by **Addr**. The command should be followed immediately by the single byte value to be written.

The following example writes the value **77** at address **2**:

```
OWOUT Owio, 0, [$E2, 77]
```

To write multiple bytes or to write to addresses beyond **7**, use the **Enter Address** command (\$EA), followed by the beginning address to write to:

1	1	1	0	1	0	1	0
0	0	0	Addr				

The address value, **Addr**, can range from **0** through **31** (**\$1F**). This address byte should be followed by one or more bytes of data, which are stored sequentially, beginning with the chosen address. This continues until a reset is received. Data received for addresses beyond **\$1F** are ignored.

The following example writes the values **123** and **64**, beginning at address **\$13**. Notice the use of the "reset after data", which is used to terminate data reception:

```
OWOUT Owio, 2, [$EA, $13, 123, 64]
```

### RAM Reads

To read a single byte from RAM, use the **Dump** command (\$Dn):

1	1	0	1	0	Addr
---	---	---	---	---	------

The number **Addr** can range from **0** to **7** and represents a shorthand notation for reading a *single byte* from the address represented by **Addr**. The command should be followed immediately by a read of a single byte.

The following example reads the value stored in location **5** and saves it in the variable **Dat**:

```
OWOUT Owio, 0, [$D5]
OWIN Owio, 0, [Dat]
```

To read multiple bytes or to read from addresses beyond **7**, use the **Dump Address** command (\$DA):

1	1	0	1	1	0	1	0
0	0	0	Addr				

The address value, **Addr**, can range from **0** through **31 (\$1F)**. This address byte should be followed by a read of one or more bytes of data, which are retrieved sequentially, beginning with the chosen address. This continues until a reset is received. Data read from addresses beyond **\$1F** are assigned the value zero.

The following example reads three values beginning at location **15**, and assigns them to the variables **I**, **J**, and **K**. Notice the use of the “reset after data” in the **OWIN** statement to terminate the read operation.

```
OWOUT Owio, 0, [$DA, 15]
OWIN Owio, 2, [I, J, K]
```

## Summary

The following table summarizes the GPIO commands:

Allowed Port No.				Command p = port no. u = pullup m = modulate	Description	Followup
3	2	1	0			
✓	✓	✓	✓	0upp0000 = \$p0	Digital Input	Read bit
✓	✓	✓	✓	0upp0001 = \$p1	Counter input, LSB	Read byte (reset before data)
✓	✓	✓	✓	0upp0010 = \$p2	Counter input, MSB	Read byte (reset before data)
✓	✓	✓	✓	0upp0011 = \$p3	Counter input, Word	Read 2 bytes (reset before data)
✓	✓	✓	✓	0mpp0100 = \$p4	Digital output 0	
✓	✓	✓	✓	0mpp0101 = \$p5	Digital output 1	
✓	✓			0mpp0110 = \$p6	PWM output positive	Output PWM value (one byte)
✓	✓			0mpp0111 = \$p7	PWM output negative	Output PWM value (one byte)
	✓	✓	✓	00pp1000 = \$p8	Compare to Port 3	Read bit
		✓	✓	00pp1001 = \$p9	ADC[7..0], Vdd ref	Read bit until 0, then read byte
		✓	✓	00pp1010 = \$pA	ADC[9..2], Vdd ref	Read bit until 0, then read byte
		✓	✓	00pp1011 = \$pB	ADC[9..0], Vdd ref	Read bit until 0, then read 2 bytes
	✓	✓	✓	00pp1100 = \$pC	Compare to 1.1V ref	Read bit
		✓	✓	00pp1101 = \$pD	ADC[7..0], 1.1V ref	Read bit until 0, then read byte
		✓	✓	00pp1110 = \$pE	ADC[9..2], 1.1V ref	Read bit until 0, then read byte
		✓	✓	00pp1111 = \$pF	ADC[9..0], 1.1V ref	Read bit until 0, then read 2 bytes
				\$F0 - \$F9	Set PWM frequency	
				\$En, n = 0-7.	Save 1 byte to RAM.	Output byte value to save.
				\$EA	Save data to RAM	Output address, data byte(s), reset
				\$Dn, n = 0-7	Read 1 byte from RAM	Read byte
				\$DA	Read data from RAM	Output address, read byte(s), reset
				\$DD	Read firmware ID	Read three bytes