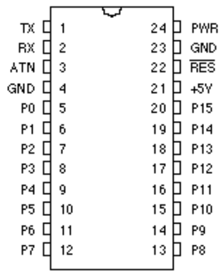
BASIC Stamp II



Thank you for purchasing the BASIC Stamp Programming Package.

This is the manual for our new BASIC Stamp II module (shown above). Everything you need to know about programming the Stamp II may be found in this manual. However, even if you're only interested in the Stamp II, you may gain useful knowledge by referring to the Stamp I manual. The Stamp I manual has a collection of interesting application notes, which may only need slight modification to run on the Stamp II.

If you have any questions, please let us know.

BLANK PAGE

BASIC Stamp II

System Requirements

To program Stamp IIs, you'll need the following computer system:

- IBM PC or compatible computer
- 3.5-inch disk drive
- Serial port
- 128K of RAM
- MS-DOS 2.0 or greater

To power the Stamps, you can use a 9-volt battery (this is the most convenient method). You can also use a 5-15 volt power supply, but you should be careful to connect the supply to the appropriate part of the Stamp. A 5-volt supply should be connected directly to the Stamp's +5V pin (also called VDD), but a higher voltage should be connected to the Stamp's PWR pin (also called VIN).

Connecting a high voltage supply (greater than 6 volts) to the 5-volt pin can permanently damage the Stamp.

Packing List

If you purchased the Stamp Programming Package, you should have received the following items:

- Stamp I programming cable (parallel port DB25-to-3 pin)
- Stamp I manual
- Stamp I application notes (contained in Stamp I manual)

- Stamp II programming cable (serial port DB9-to-DB9)
- Stamp II manual (this booklet)

- 3.5-inch diskette

If any items are missing, please let us know.

Connecting to the PC

To program a Stamp II, you'll need to connect it to your PC and then run the editor / downloader software. In this booklet, it's assumed that you have a BS2-IC and its corresponding carrier board (shown below).

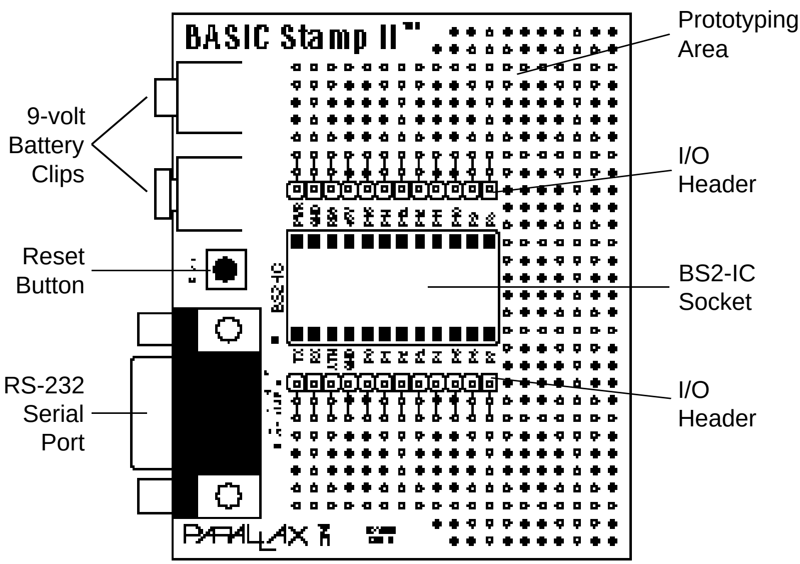
To connect the Stamp II to your PC, follow these steps:

1) Plug the BS2-IC onto the carrier board. The BS2-IC plugs into a 24-pin DIP socket, located in the center of the carrier. When plugged onto the carrier board, the words "Parallax BS2-IC" should be near the reset button.

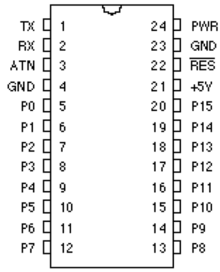
2) In the Stamp Programming Package, you received a serial cable to connect the Stamp II to your PC. Plug the female end into an available serial port on your PC.

3) Plug the male end of the serial cable into the carrier board's serial port.

4) Supply power to the carrier board, either by connecting a 9-volt battery or by providing an external power source.



BASIC Stamp II



Pin	Name	Description	Comments
1	TX	Serial output	Connect to pin 2 of PC serial DB9 (RX) *
2	RX	Serial input	Connect to pin 3 of PC serial DB9 (TX) *
3	ATN	Active-high reset	Connect to pin 4 of PC serial DB9 (DTR) *
4	GND	Serial ground	Connect to pin 5 of PC serial DB9 (GND) *
5	P0	I/O pin 0	Each pin can source 20 ma and sink 25 ma.
6	P1	I/O pin 1	
7	P2	I/O pin 2	P0-P7 and P8-P15, as groups, can each
8	P3	I/O pin 3	source a total of 40 ma and sink 50 ma.
9	P4	I/O pin 4	
10	P5	I/O pin 5	
11	P6	I/O pin 6	
12	P7	I/O pin 7	
13	P8	I/O pin 8	
14	P9	I/O pin 9	
15	P10	I/O pin 10	
16	P11	I/O pin 11	
17	P12	I/O pin 12	
18	P13	I/O pin 13	
19	P14	I/O pin 14	
20	P15	I/O pin 15	
21	+5V **	+5V supply	5-volt input or regulated output.
22	RES	Active-low reset	Pull low to reset; goes low during reset.
23	GND	System ground	
24	PWR **	Regulator input	Voltage regulator input; takes 5-15 volts.

* For automatic serial port selection by the Stamp II software, there must also be a connection from DSR (DB9 pin 6) to RTS (DB9 pin 7). This connection is made on the Stamp II carrier board. If you are not using the carrier board, then you must make this connection yourself, or use the command-line option to tell the software which serial port to use.

** During normal operation, the Stamp II takes about 7 mA. In various power-down modes, consumption can be reduced to about 50 µA.

Starting the Editor

With the Stamp II connected and powered, insert the BASIC Stamp diskette and then enter the Stamp II directory by typing the following command from the DOS prompt:

```
CD STAMP2
```

Once in the Stamp II directory, you can run the Stamp II editor/ downloader software by typing the following command:

```
STAMP2
```

The Stamp II software will start running after several seconds. The editor screen is dark blue, with one line across the top that indicates how to get on-screen editor help. Except for the top line, the entire screen is available for entering and editing BASIC programs.

Command-line options:

There are several command-line options that may be useful when running the software; these options are shown below:

STAMP2 *filename*	Runs the editor and loads *filename.*
STAMP2 /m	Runs the editor in monochrome mode. May give a better display on some systems, especially laptop computers.
STAMP2 /*n*	Runs the editor and specifies which serial port to use when downloading to the Stamp II (note that *n* must be replaced with a serial port number of 1-4).

Normally, the software finds the Stamp II by looking on all serial ports for a connection between DSR and RTS (this connection is made on the carrier board). If the DSR-RTS connection is not present, then you must tell the software which port to use, as shown above.

Entering & Editing Programs

We've tried to make the editor as intuitive as possible: to move up, press the *up arrow*; to highlight one character to the right, press *shift-right arrow*; etc.

Most functions of the editor are easy to use. Using single keystrokes, you can perform the following common functions:

- Load, save, and run programs.
- Move the cursor in increments of one character, one word, one line, one screen, or to the beginning or end of a file.
- Highlight text in blocks of one character, one word, one line, one screen, or to the beginning or end of a file.
- Cut, copy, and paste highlighted text.
- Search for and / or replace text.
- See how the Stamp IIs memory is being used by your program.
- Identify the version of the BASIC interpreter in your Stamp II.

Editor Function Keys

The following list shows the keys that are used to perform various functions:

F1	Display editor help screen.
Alt-R	Run program in Stamp II *(download the program on the screen, then run it)*
Alt-L	Load program from disk
Alt-S	Save program on disk
Alt-M	Show memory usage maps
Alt-I	Show version number of BASIC interpreter
Alt-Q	Quit editor and return to DOS
Enter	Enter information and move down one line
Tab	Same as Enter

Left arrow	Move left one character
Right arrow	Move right one character
Up arrow	Move up one line
Down arrow	Move down one line
Ctrl-Left	Move left to next word
Ctrl-Right	Move right to next word
Home	Move to beginning of line
End	Move to end of line
Page Up	Move up one screen
Page Down	Move down one screen
Ctrl-Page Up	Move to beginning of file
Ctrl-Page Down	Move to end of file
Shift-Left	Highlight one character to the left
Shift-Right	Highlight one character to the right
Shift-Up	Highlight one line up
Shift-Down	Highlight one line down
Shift-Ctrl-Left	Highlight one word to the left
Shift-Ctrl-Right	Highlight one word to the right
Shift-Home	Highlight to beginning of line
Shift-End	Highlight to end of line
Shift-Page Up	Highlight one screen up
Shift-Page Down	Highlight one screen down
Shift-Ctrl-Page Up	Highlight to beginning of file
Shift-Ctrl-Page Down	Highlight to end of file
Shift-Insert	Highlight word at cursor
ESC	Cancel highlighted text
Backspace	Delete one character to the left
Delete	Delete character at cursor
Shift-Backspace	Delete from left character to beginning of line
Shift-Delete	Delete to end of line
Ctrl-Backspace	Delete line
Alt-X	Cut marked text and place in clipboard
Alt-C	Copy marked text to clipboard
Alt-V	Paste (insert) clipboard text at cursor
Alt-F	Find string (establish search information)
Alt-N	Find next occurrence of string

BASIC Stamp II

The following list is a summary of the BASIC instructions used by the BASIC Stamp II. Later in this pamphlet, you'll find complete descriptions of each instruction.

◆ This symbol indicates new or greatly improved instructions (compared to the BASIC Stamp I).

BRANCHING

IF...THEN	Compare and conditionally branch.
BRANCH	Branch to address specified by offset.
GOTO	Branch to address.
GOSUB	Branch to subroutine at address. GOSUBs may be nested up to four levels deep, and you may have up to 255 GOSUBs in your program.
RETURN	Return from subroutine.

LOOPING

FOR...NEXT	Establish a FOR-NEXT loop.

NUMERICS

LOOKUP	Lookup data specified by offset and store in variable. This instruction provides a means to make a lookup table.
LOOKDOWN	Find target's match number (0-N) and store in variable.
RANDOM	Generate a pseudo-random number.

DIGITAL I/O

INPUT	Make pin an input
OUTPUT	Make pin an output.
REVERSE	If pin is an output, make it an input. If pin is an input, make it an output.
LOW	Make pin output low.
HIGH	Make pin output high.
TOGGLE	Make pin an output and toggle state.
PULSIN	Measure an input pulse (resolution of 2 µs).

	PULSOUT	Output a timed pulse by inverting a pin for some time (resolution of 2 µs).
	BUTTON	Debounce button, perform auto-repeat, and branch to address if button is in target state.
◆	SHIFTIN	Shift bits in from parallel-to-serial shift register.
◆	SHIFTOUT	Shift bits out to serial-to-parallel shift register.
◆	COUNT	Count cycles on a pin for a given amount of time (0 - 125 kHz, assuming a 50/50 duty cycle).
◆	XOUT	Generate X-10 powerline control codes. For use with TW523 or TW513 powerline interface module.

SERIAL I/O

◆	SERIN	Serial input with optional qualifiers, time-out, and flow control. If qualifiers are given, then the instruction will wait until they are received before filling variables or continuing to the next instruction. If a time-out value is given, then the instruction will abort after receiving nothing for a given amount of time. Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data received must be N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).
◆	SEROUT	Send data serially with optional byte pacing and flow control. If a pace value is given, then the instruction will insert a specified delay between each byte sent (pacing is not available with flow control). Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data is sent as N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).

ANALOG I/O

	PWM	Output PWM, then return pin to input. This can be used to output analog voltages (0-5V) using a capacitor and resistor.
◆	RCTIME	Measure an RC charge/discharge time. Can be used to measure potentiometers.

SOUND

◆	FREQOUT	Generate one or two sinewaves of specified frequencies (each from 0 - 32767 hz.).
◆	DTMFOUT	Generate DTMF telephone tones.

EEPROM ACCESS

◆	DATA	Store data in EEPROM before downloading BASIC program.
	READ	Read EEPROM byte into variable.
	WRITE	Write byte into EEPROM.

TIME

PAUSE	Pause execution for 0–65535 milliseconds.

POWER CONTROL

NAP	Nap for a short period. Power consumption is reduced.
SLEEP	Sleep for 1-65535 seconds. Power consumption is reduced to approximately 50 µA.
END	Sleep until the power cycles or the PC connects. Power consumption is reduced to approximately 50 µA.

PROGRAM DEBUGGING

DEBUG	Send variables to PC for viewing.

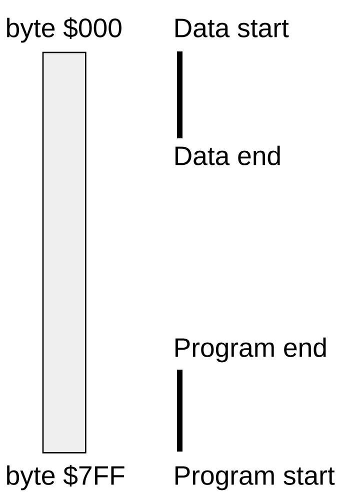
Program and Data Memory

The BS2 has 2K bytes of EEPROM which holds the executable BASIC program and any data. Memory not used by the BASIC program can be read and written at run-time as a data bank, or initialized with data at download time. This memory is only affected by downloading or run-time modification.

There are 32 bytes of RAM which serve as variable space and I/O pin interface for the BASIC program. This memory can be accessed as words, bytes, nibbles, or bits. Each time the BASIC program is run anew, this memory is cleared to all zeroes.

So, the 2K byte EEPROM is for program and data, and only affected by initial downloading or run-time modification. It survives power-down. The 32 bytes of RAM are for run-time variables and I/O pin access. This memory is cleared each time the BS2 is powered up, reset, or downloaded to.

The 2K-byte EEPROM is arranged as follows:



The 32-byte RAM is arranged as follows:

Word	Bits	Description	R/W
$0	0000 0000 0000 0000	Pin input states	read-only
$1	0000 0000 0000 0000	Pin output latches	read/write
$2	0000 0000 0000 0000	Pin directions	read/write
$3	0000 0000 0000 0000	variable space	read/write
$4	0000 0000 0000 0000	variable space	read/write
$5	0000 0000 0000 0000	variable space	read/write
$6	0000 0000 0000 0000	variable space	read/write
$7	0000 0000 0000 0000	variable space	read/write
$8	0000 0000 0000 0000	variable space	read/write
$9	0000 0000 0000 0000	variable space	read/write
$A	0000 0000 0000 0000	variable space	read/write
$B	0000 0000 0000 0000	variable space	read/write
$C	0000 0000 0000 0000	variable space	read/write
$D	0000 0000 0000 0000	variable space	read/write
$E	0000 0000 0000 0000	variable space	read/write
$F	0000 0000 0000 0000	variable space	read/write

Word $0 always reflects the read-state of all 16 I/O pins. Whether a pin is an input or output, it's logical state can be read in this word. Word $0 is accessed by the following symbolic names:

INS	the entire 16-bit word
INL	the low byte of INS
INH	the high byte of INS
INA	the low nibble of INL
INB	the high nibble of INL
INC	the low nibble of INH
IND	the high nibble of INH
IN0	the low bit of INS (corresponds to I/O pin P0)
|	|
IN15	the high bit of INS (corresponds to I/O pin P15)

Word $1 contains the output latches for all 16 I/O pins. If a pin is in input mode, this data is unused; but, when a pin is in output mode, its corresponding word $1 bit sets its state. The bits are all readable and writable, regardless of pin direction. These are its symbolic names:

OUTS	the entire 16-bit word
OUTL	the low byte of OUTS
OUTH	the high byte of OUTS
OUTA	the low nibble of OUTL
OUTB	the high nibble of OUTL
OUTC	the low nibble of OUTH
OUTD	the high nibble of OUTH
OUT0	the low bit of OUTS - corresponds to pin P0
|	
OUT15	the high bit of OUTS - corresponds to pin p15

Word $2 contains the direction bits for all 16 I/O pins. To place a pin in input mode, its corresponding word $2 bit must be cleared to 0. To place a pin into output mode, its corresponding word $2 bit must be set to 1, at which time its word $1 bit will determine whether it drives high or low. Word $2 has these symbolic names:

DIRS	the entire 16-bit word
DIRL	the low byte of DIRS
DIRH	the high byte of DIRS
DIRA	the low nibble of DIRL
DIRB	the high nibble of DIRL
DIRC	the low nibble of DIRH
DIRD	the high nibble of DIRH
DIR0	the low bit of DIRS - corresponds to pin P0
|	
DIR15	the high bit of DIRS - corresponds to pin p15

Words $3-$F are for general purpose variable use and have no pre-assigned symbolic names. The VAR statement is used to allocate this memory.

The above text introduced the physical pin-out of the BASIC Stamp II, as well as the internal EEPROM, RAM, and I/O structure. The following text discusses the programming of the BS2.

Programming the BASIC Stamp II

In the BASIC Stamp II, there are two general categories of BASIC statements: compile-time and run-time.

> Compile-time statements are resolved when you compile the program (Alt-R or Alt-M), and they do not generate any executable code.
>
> Run-time statements generate code and are executed at run-time.

There are three compile-time statements. They are used for declaring variables, constants, and data. They are:

> VAR, CON, and DATA

The VAR statment - defining variables

Your program should begin with a declaration of all of its variables. VAR statements assign symbolic names to variable RAM (RAM not used by I/O - words $3-$F). This is done as follows:

```
'Declare the variables

cat     var  nib      'make "cat" a nibble variable
mouse   var  bit      'make "mouse" a bit variable
dog     var  byte     'make "dog" a byte variable
rhino   var  word     'make "rhino" a word variable
snake   var  bit(10)  'make "snake" a 10-piece bit variable
```

The compiler will group all words, bytes, nibs, and bits, and respectively arrange them into unused RAM. By pressing Alt-M, you can see a picture of the RAM allocation. First, the three I/O words are shown, then all words, bytes, nibs, and finally, bits, are seen. Empty RAM follows. Alt-M is a quick way to assess how much RAM you've used.

The VAR usage options are as follows:

```
'define unique variables

sym1    VAR  bit               'make a bit variable
sym2    VAR  nib               'make a nibble variable
sym3    VAR  byte              'make a byte variable
sym4    VAR  word              'make a word variable

'After bit/nib/byte/word a value may be placed
'within parentheses to declare an array size:

sym5    VAR  nib (10)          'make a 10 nibble array

'define variables-within-variables or alias variables

sym6    VAR  sym4.highbit      'make a bit variable of sym4's highbit
sym7    VAR  sym4.lowbit       'make a bit variable of sym4's lowbit
sym8    VAR  sym2              'make an alternate name for sym2

'When using VAR to assign non-unique variables (a variable
'name is used in lieu of bit/nib/byte/word(size)), a period may
'be placed after the variable name and followed by modifiers.
'Modifiers are used to identify sub-pieces of the initially-
'mentioned variable.

sym9    VAR  sym4.highbyte.lownib.bit2     'picky, picky...
```

Here are all the variable modifiers:

```
LOWBYTE         'low byte of a word
HIGHBYTE        'high byte of a word
BYTE0           'byte0 (low byte) of a word
BYTE1           'byte1 (high byte) of a word
```

```
LOWNIB      'low nibble of a word or byte
HIGHNIB     'high nibble of a word or byte
NIB0        'nib0 of a word or byte
NIB1        'nib1 of a word or byte
NIB2        'nib2 of a word
NIB3        'nib3 of a word

LOWBIT      'low bit of a word, byte, or nibble
HIGHBIT     'high bit of a word, byte, or nibble
BIT0        'bit0 of a word, byte, or nibble
BIT1        'bit1 of a word, byte, or nibble
BIT2        'bit2 of a word, byte, or nibble
BIT3        'bit3 of a word, byte, or nibble
BIT4        'bit4 of a word or byte
BIT5        'bit5 of a word or byte
BIT6        'bit6 of a word or byte
BIT7        'bit7 of a word or byte
BIT8        'bit8 of a word
BIT9        'bit9 of a word
BIT10       'bit10 of a word
BIT11       'bit11 of a word
BIT12       'bit12 of a word
BIT13       'bit13 of a word
BIT14       'bit14 of a word
BIT15       'bit15 of a word
```

In summary, to declare variables, VAR statements are used. VAR statements either declare unique variables or variables-within-variables and alias-variables.

For defining unique variables:

```
symbol      VAR     size (array)
```

- symbol is a unique name for a variable
- size is either WORD, BYTE, NIB, or BIT
- (array) is an optional expression which declares an array size

For defining variables-within-variables or alias-variables:

```
symbol    VAR   variable.modifiers
```

- symbol is a unique name for a variable
- variable is a defined variable name
- .modifiers are optional and used to define variables-within-variables

The compiler will group all declarations by size (in the case of unique variables) and assign them to unused RAM. Alt-M lets you see the result of this process. Non-unique variables are in-whole or in-part derived from unique variables and get assigned within the unique-variable memory.

Keep in mind that you may make alias names for the pin variables:

```
keyin     var   in5            'make "keyin" a way to read P5's state.
```

Note for Stamp I users: *W0-W12 (and the corresponding B0-B25) are pre-defined by the compiler software to make use of Stamp I programs easier. If you use a Stamp I program with the Stamp II, you can enter the older "Wx" and "Bx" variable names without having to define them first.*

The CON statment - defining constants

The CON statement is similar to the VAR statement, except that it is for defining constant values and assigning them to symbolic names. This is handy for having a single declaration which gets accessed throughout your program. The CON syntax is as follows:

```
symbol    CON   expression     'assign expression to "symbol"
```

- symbol is a unique symbolic name for a constant
- expression is a compile-time-resolvable constant

```
level     CON   10             '"level" is same as 10 in program
limit     CON   10*4<<2        '"limit" is 160
```

expressions after CON can contain the following binary operators and are resolved left-to-right:

```
+     add
-     subtract
*     multiply
/     divide
<<    shift left
>>    shift right
&     logical AND
|     logical OR
^     logical XOR
```

example:

```
growth    CON   100-light/gel    '"light" and "gel" are CON's, too
```

The DATA statment - defining data

EEPROM memory not used by your BASIC program can be used for data storage. Keep in mind that your BASIC program builds from the end of memory towards the start of memory. This allocation is automatic. Your data, on the other hand, builds from the start of memory towards the end. The sum of program and data memory cannot exceed the 2K byte limit. The compiler will always tell you when you have a conflict.

DATA statements are used to build data into unused memory. Initially, the DATA location is set to 0. It is advanced by 1 for each byte declared. Here is an example DATA statement:

```
table   DATA     "Here is a string..."
```

Usually, you'll want to precede DATA statements with a unique symbol. The symbol will be assigned a constant value (as if via CON) which is the current data pointer. The text following 'DATA' is usually a list of bytes which can be constant expressions. In the above example (assuming this was the first DATA statement in the program), "table" becomes a constant symbol of value 0; "Here is a string..." is broken into individual bytes and placed into EEPROM memory sequentially. Alt-M and two <SPACE>s will show you the result of this line.

The DATA pointer may be altered at any time by an @ sign followed by a new pointer value:

```
list    DATA    @$100,"some data"
```

DATA has a few variations of use to allocate defined and undefined data. Defined data is fully declared and known at compile time. Undefined data is the mere allocation of data space, while not assigning values into the bytes of EEPROM (to be done at run-time, instead). Defined and undefined data are declared as follows:

for defined data:

```
fee     DATA    0,1,2,3,4,5,6,7,8,9    'actual bytes
fie     DATA    word 1000              'make two bytes: $E8 and $03
foe     DATA    0 (256)                '256 bytes initialized as 0
```

for undefined data:

```
fum     DATA    (1024)                 'reserved 1K byte of undefined data
abc     DATA    word (16)              'reserve 16 words of undefined data
```

Important concept: Defined DATA and BASIC program memory are always downloaded to the BS2. Undefined data and unused EEPROM memory are not downloaded. This allows you to change programs while keeping data, assuming both programs defined the same stretch of memory as undefined DATA. Alt-M will show you maps of EEPROM allocation. This download / don't-download rule is applied to 16-byte blocks. If any byte within a 16-byte block is defined DATA or BASIC program, that whole block is downloaded. Use Alt-M to see this.

In summary, DATA is used to define EEPROM byte usage that doesn't conflict with the BASIC program storage:

- DATA can be preceeded by a symbol which will be assigned the constant value of the current DATA pointer.
- Byte-size data is assumed, but 'word' can be used to break a word into two bytes of storage.
- The @ sign is used to redirect the DATA pointer. If a symbol preceeds

a DATA statement and the first thing after DATA is @, the new pointer value is assigned to the symbol.

- Defined data is spelled out, so to speak, with numbers and letters.
- Defined data may be repeated at the byte or word level using (array).
- Undefined data may be reserved by using (array) unpreceeded by a value.

Note: DATA can contain references to DATA symbols:

```
t1      DATA    "Here's table 1...",0
t2      DATA    "Here's table 2...",0
t3      DATA    "Here's table 3...",0
t4      DATA    "Here's table 4...",0

start   DATA    word t1, word t2, word t3, word t4
```

Run-Time Expressions

Run-time expressions can contain constants, variables, operators, and parentheses. They are resolved using 16-bit math.

Constants can be in several forms:

label	A label may be assigned a constant via CON.
$BA1F	Hex
%111001111	Binary
99	Decimal
"A"	ASCII

Note: When more than one character is within quotes, they are separated by the compiler as such: "DOG" becomes "D","O","G". "String"+$80 becomes: "S","t","r","i","n","g"+$80.

Variables can be accessed a number of ways:

somevar	Some variable.
wordvar.highbit	Use modifiers to access sub-variables.

nibarray(index)	A variable followed by an expression in quotes is indexed as an array (0=1st element).
word.bit0(bitoffset)	Scan a word, one bit at a time.

If a variable were defined as:

string var byte (10)

It could be accessed as:

string	The 1st byte
string (0)	The 1st byte
string (1)	The 2nd byte
string (9)	The 10th (last) byte
string.lownib(nibindex)	Nibindex could be 0-19
string.lowbit(bitindex)	Bitindex could be 0-79

There are also binary, unary, and conditional expression operators.

Unary operators preceed a variable or constant or (expression) and have highest priority. They are as follows:

SQR	Square root of unsigned 16-bit value
ABS	Absolute of signed 16-bit value
~	One's complement of 16-bit value (bitwise not)
-	Two's complement of 16-bit value (negation)
DCD	2^n decoder of 4-bit value (0...15 -> 1,2,4,8,16,...32768)
NCD	Priority encoder of 16-bit value (=>32768,=>16384,=>8192,...=1 -> 15,14,13,...1 ; 0 -> $FFFF)
COS	Cosine of 8-bit value. Result is in the range of +-127, unit circle is 0-255 radial units.
SIN	Sine of 8-bit value. Result is in the range of +-127, unit circle is 0-255 radial units.

Examples:

sin bytevar

```
sqr 50000
~ in0
```

Binary operators take two terms and go between variables, constants, or expressions. They are as follows:

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
MIN	Limit value to minimum
MAX	Limit value to maximum
+	Addition
-	Subtraction
*	Multiply
**	Multiply and return high 16-bits of result
*/	Multiply and return middle 16-bits of result (use to simultaneously multiply by a whole and a part; ie. ‘value */ $0180’ multiplies by 1 and a half)
/	Divide
//	Divide and return remainder
DIG	Return decimal digit; ‘12345 dig 3’ returns 2
<<	Shift left
>>	Shift right
REV	Reverse order of bits, lsb-justified; ‘%100110 rev 6’ yields %011001

Examples:

```
ypos * xsize + xpos
randword // 20
countacc min 200 - 200 / 200
```

Parentheses can be placed to special-order the pattern of expression resolution. Though unary operators have highest priority and binary operators have secondary priority, and with those rules expressions are resolved left-to-right, parentheses can override priority:

```
X+1*Y-1          'something's wrong here if we need (X+1)*(Y-1)
(X+1)*(Y-1)      'do it right
```

Up to 8 levels of parentheses can be used.

For use within conditional expressions (IF), there is a special unary operator and several binary operators. These conditional operators have highest priority of all.

NOT	Highest priority unary
AND, OR, XOR	Highest priority binaries

Note: These are arithmetically identical to expression operators:

~
&
|
^

though they differ in application.

Lower-priority conditional binary operators (still higher than expression ops):

<	Less than
<=	Less than or equal to
=	Equal to
=>	Equal to or greater than
>	Greater than
<>	Not equal

Note: These comparison operators return 0 for false and $FFFF for true. Combined with NOT, AND, OR, and XOR, complex tests can be done.

To summarize, here are some examples:

```
outs = ~ dcd nibarray(index)       'lookup a nibble, decode it, not it
IF x<1 or not y>3 and (z=0 xor r=3) then loopback
```

BRANCH

Branch according to an index.

usage: BRANCH index,[label0,label1,label2,...labelN]

If *index*=0, a GOTO label0 will be executed. If *index*=1, a GOTO label1 will be executed. If the index exceeds the number of label entries, no branch will occur and execution will proceed at the next instruction.

BUTTON

Debounce button, auto-repeat, and branch if button is in target state.

usage: BUTTON pin,downstate,delay,rate,bytevariable,targetstate,label

Pin will be placed in input mode.

Downstate is the state which is read when the button is pressed.

Delay specifies down-time before auto-repeat in BUTTON cycles.

Rate specifies the auto-repeat rate in BUTTON cycles.

Bytevariable is workspace for the BUTTON instruction. It must be cleared to 0 before being used by BUTTON for the first time.

Targetstate specifies what state (0=not pressed, 1=pressed) the button should be in for a branch to occur.

Label specifies where to go if the button is in the target state.

Please refer to the Stamp I manual for an in-depth description of the BUTTON instruction.

COUNT

Count cycles on a pin for some milliseconds.

usage: COUNT pin,period,wordvariable

Pin will be placed in input mode. For *period* milliseconds, cycles will be counted on the designated I/O pin. Both sides of the waveform must be at least 4µs in duration, which limits the input frequency to 125-kHz (assuming a 50/50 duty cycle). The result (0-65535) will be written into *wordvariable.*

Note that a word variable (16 bits) must be used to count above 255. If you only need values between 0-255, then a byte variable will work.

DEBUG outputdata

Show variables and messages for debugging purposes.

usage:	DEBUG "Here we are!"	Show message when executed

When executed, the data after DEBUG will be sent to the PC for display. DEBUG data can be displayed in several modes. Straight data can be relayed to the PC, or you can have values printed in decimal, hex, binary, or ASCII. In the number-printing modes, the result of an expression can be printed or a complete relation can be shown between the expression and its result. For example:

DEBUG dec? X	Show X in decimal with relation

will yield (if x=1):

```
X = 1
```

DEBUG dec X	Show X in decimal

yields:

```
1
```

When printing numbers, these prefixes can be used before expressions:

ASC?		Show ASCII value with relation
STR	bytevar	Output string from byte array - until 0
STR	bytevar\n	Output string from byte array - n bytes
REP	value\n	Output value n times
DEC	value	Print value in decimal
DEC1-DEC5	value	Print value in decimal - 1-5 digits
SDEC	value	Print value in signed decimal
SDEC1-SDEC5	value	Print value in signed decimal - 1-5 digits
HEX	value	Print value in hex
HEX1-HEX4	value	Print value in hex - 1-4 digits
SHEX	value	Print value in signed hex
SHEX1-SHEX4	value	Print value in signed hex - 1-4 digits
IHEX	value	Print value in hex w/'$'
IHEX1-IHEX4	value	Print value in hex w/'$' - 1-4 digits

ISHEX	value	Print value in signed hex w/'$'
ISHEX1-ISHEX4	value	Print value in signed hex w/'$' - 1-4 digits
BIN	value	Print value in binary
BIN1-BIN4	value	Print value in binary - 1-4 digits
SBIN	value	Print value in signed binary
SBIN1-SBIN16	value	Print value in signed binary - 1-4 digits
IBIN	value	Print value in binary w/'%'
IBIN1-IBIN16	value	Print value in binary w/'%' - 1-4 digits
ISBIN	value	Print value in signed binary w/'%'
ISBIN1-ISBIN16	value	Print value in signed binary w/'%' -1-4 digits

REP cannot be followed by a '?', but ASC needs one.

DEBUG statements can contain many strings and numbers, separated by commas. In addition, there are several special control characters which are interpreted by the PC:

Name	Value	Effect
CLS	0	Clears the screen and homes the cursor
HOME	1	Homes the cursor
BELL	7	Beep the PC speaker
BKSP	8	Backspace - backs up the cursor
TAB	9	Advances to the next 8th column
CR	13	Carriage return - down to next line

Example program using DEBUG:

```
count   var     byte                    'Define a byte variable called count

loop1:  debug cls, bell                 'Clear the screen and beep the speaker

loop2:  debug sdec? sin count           'Show the signed-decimal sine of count
        count = count + 1               'Increment count
        if count <> 0 then loop2        'Loop until count rolls over

        pause 1000                      'Pause for 1 second
        goto loop1                      'Repeat the program
```

DTMFOUT

Output DTMF tones.

usage: DTMFOUT pin,{ontime,offtime,}[key,key,key,...]

Pin will be temporarily placed in output mode for modulation. The default on and off times are 200 ms and 50 ms.

Ontime and *offtime* are optional overrides in milliseconds.

Key(s) are 0-15, which are the following tones:

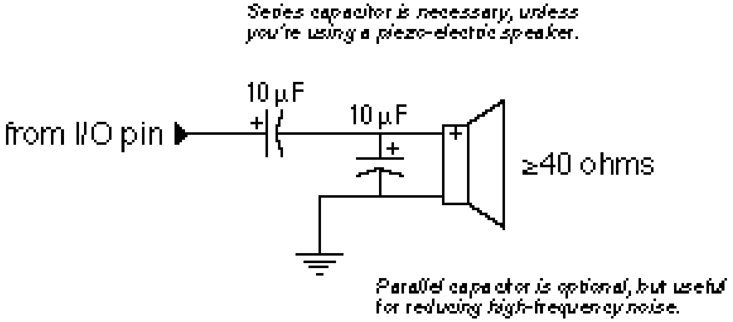
0-9 are the digits 0-9

10 is *

11 is #

12-15 are A-D, which are fourth-column codes unavailable on normal telephones.

The pin may be filtered by an RC circuit to achieve a clean sine-wave. High-impedance speakers may be driven with a coupling cap and a filter cap (see below).



END

End program.

usage: END

Enter low-power mode and keep I/O's updated. Every ~2.3 seconds the I/O's will go high-z for ~18ms. Approximately 50µa average current will be consumed.

END is terminated only by a hardware reset.

FOR...NEXT

Establish a FOR...NEXT loop.

```
usage:    FOR variable = start TO end {STEP stepval}
          {some code}
          NEXT
```

STEP is for specifying a step value other than the default of 1; if specified, stepval must be positive since whether to add or subtract stepval from variable is determined dynamically at run-time (this allows '10 TO 0' without specifying a negative STEP value.).

FOR...NEXT loops can be nested up to 16 deep.

Note: NEXT is stand-alone and implies the variable from the last FOR statement.

FREQOUT

Output a sine-wave(s) for some time.

usage: FREQOUT pin,milliseconds,freq1{,freq2}

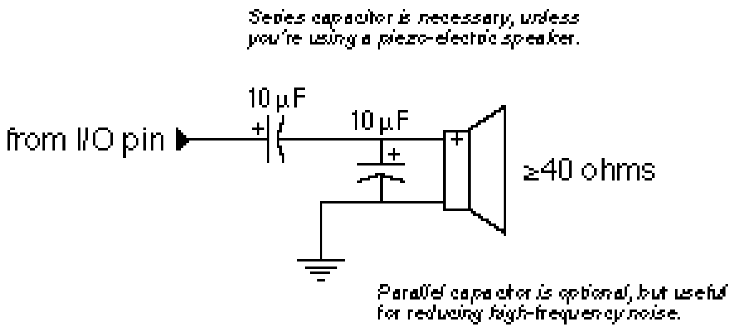
Pin will be temporarily placed in output mode for modulation.

Milliseconds specifies how long to output the frequency(s).

Freq1 specifies the (first) frequency in 1Hz units (0-32768 Hz).

Freq2 is optional and specifies a second frequency.

The pin may be filtered by an RC circuit to achieve a clean sine-wave. High-impedance speakers may be driven with a coupling cap and a filter cap (see below).



GOSUB

Go to a subroutine (and then RETURN later).

usage: GOSUB *ledset*

The execution point is stored and then a branch to *ledset* occurs. When a RETURN is encountered (in *ledset*), execution continues at the instruction following the GOSUB.

GOSUB's may be nested up to 4 deep and you are allowed 255 of them in your program.

GOTO

Go to a new point in the program.

usage: GOTO joe

A branch to *joe* will occur, rather than execution continuing at the next instruction.

HIGH

Make a pin an output and have it output 1.

usage: HIGH pin

Pin is 0-15.

Make the pin an output, and change its output latch to 1.

IF...THEN

Branch conditionally.

usage: IF conditionalexpression THEN label
ie. IF x=1 then redoit

If the result of conditionalexpression is not 0, execution will be continued at *label.* Otherwise, execution continues at the next instruction.

INPUT

Make a pin an input.

usage: INPUT pin

Pin is 0-15.

Make a pin an input. The pin's output latch is unaffected.

LOOKDOWN

Lookdown a value and return an index.

usage: LOOKDOWN value,??[value0,value1,value2,...valueN],variable

?? is a comparison operator: =,<>,>,<,<=,=> (= is the default).

A comparison is made between *value* and *value0*; if the result is true, 0 is written into *variable.* If that comparison was false, another comparison is made between *value* and *value1*; if the result is true, 1 is written into variable. This process continues until a true is yielded, at which time the index is written into *variable*, or until all entries are exhausted, in which case *variable* is unaffected.

LOOKUP

Lookup a variable according to an index.

usage: LOOKUP index,[value0,value1,value2,...valueN],variable

If *index*=0, then *value0* will be written to *variable.* If *index*=1, then *value1* will be written to *variable.* If the index exceeds the number of value entries, then *variable* will not be affected.

LOW

Make a pin an output and have it output 0.

usage: LOW pin

Pin is 0-15.

Make the pin an output, and change its output latch to 0.

NAP

Nap for a short period.

usage: NAP x

Enter low-power mode for a short period. When the period is over, the I/O's will go high-z for ~18ms and execution will continue at the next instruction.

The x values for NAP are as follows:

x	~seconds
0	0.018
1	0.036
2	0.072
3	0.140
4	0.290
5	0.580
6	1.200
7	2.300

OUTPUT

Make a pin an output.

usage: OUTPUT pin

Pin is 0-15.

The pin will be made an output. The pin's output latch is not affected.

PAUSE

Pause for x milliseconds (x=0 to 65535).

usage: PAUSE x

A delay of x milliseconds will occur.

PULSIN

Input a timed pulse.

usage: PULSIN pin,state,variable

Pin will be placed in input mode. A pulse of *state* will be awaited and measured with 2µs resolution, and the result will be written into *variable.*

If an overflow occurs while waiting for any edge, (>65535*2µs or >131ms), 0 will be written into variable.

PULSOUT

Output a timed pulse.

usage: PULSOUT pin,period

Pin will be made an output opposite of it's OUTx value for *period**2µs. *Pin* will be left in output mode with OUTx state.

Period may be a value from 0-65535; a value of 0 will result in no output pulse.

PWM

Pulse-width modulate a pin for some time.

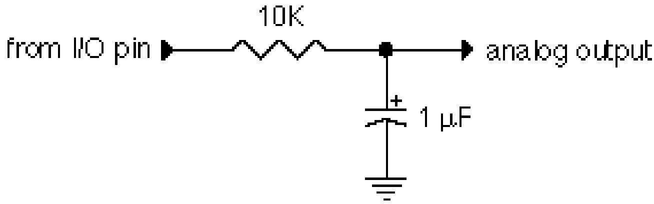
usage: PWM pin,duty,cycles

Pin will be temporarily made an output while it is modulated.

Duty is an 8-bit value (0-255) which specifies the duty-cycle.

Cycles is the number of 256 pin update periods which take ~1ms. When done, *pin* will be placed in input mode to allow the voltage to remain until another PWM is executed.

A digital-to-analog converter can be made by connecting the pin to a resistor which goes to a cap which goes to GND. The resistor-cap junction will reflect the duty (0-5V) during and after PWM (see below).



RANDOM

Pseudo-randomly iterate a word variable.

usage: RANDOM wordvariable

wordvariable will be iterated 16 times to potentially change every bit.

RCTIME

Measure an RC charge/discharge time.

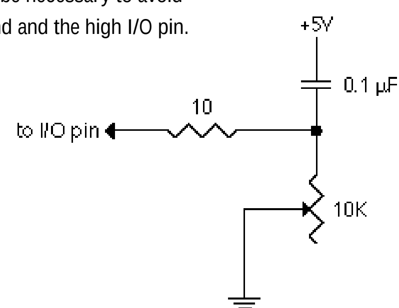
usage: RCTIME pin,state,variable

Pin will be placed in input mode and time will be measured in 2µs units while *pin* is in *state.* The result will be written into *variable.* If an overflow occurs (>131ms), 0 will be written.

This is useful for measuring resistor-capacitor charge / discharge times. Since the logic threshold of a pin is ~1.4 volts, it is best to have the RC voltage go from the 5V supply towards ground, yielding a span of ~3.6 volts. For example, to convert a potentiometer setting to a number:

> Connect the pot as shown below. Make the pin high for 1 ms, then do RCTIME. The resistor-cap junction will start at 5V and fall towards ground. When ~1.4 is reached, the counting will terminate and the result will be returned to *variable.* Adjusting the pot causes the returned value to go up or down.

Note: the 10-ohm resistor may be necessary to avoid a possible short between ground and the high I/O pin.



Sample Program:

```
Pot    var     word

Loop:  HIGH 0              'Discharge capacitor
       PAUSE 1             'Allow time for discharge
       RCTIME 0,1,Pot      'Measure chrage time
       DEBUG ?Pot          'Show result on PC
       GOTO loop           'Repeat the process
```

READ

Read a byte from the EEPROM.

usage: READ location,variable

Location is 0-2047. *Variable* will receive the byte read from *location.*

RETURN

Return from a subroutine.

usage: RETURN

The execution point is set to the instruction after the GOSUB that got into the subroutine executing the RETURN.

If a RETURN is executed without a corresponding GOSUB, execution begins anew at the start of the program.

REVERSE

Reverse a pin's input/output direction.

usage: REVERSE pin

Pin is 0-15.

The pin will be changed from an input to an output, or from an output to an input. The state of its output latch is not affected.

SERIN rpin{\fpin},baudmode,{plabel,}{timeout,tlabel,}[inputdata]

Input data serially.

---

example:

```
SERIN 4,32+$4000,60000,nothingcame,[WAIT (“,”),STR bytearray\16\”,”]
```

Serial pin
Baudrate, etc.
Timeout delay
Jump to if timeout
Wait for comma
Receive 16 bytes

Accept serial data on pin 4. Reception mode is 19,200 baud, 8 data bits, no parity, inverted data. If no data is received after 60 seconds (60,000 milliseconds), execution will jump to *nothingcame.* If data is received, SERIN will wait for a comma (“,”) before accepting bytes. After the comma is received, up to 16 bytes will be received and placed into *bytearray;* if another comma is received before 16 bytes of data, then the remaining bytes in the array will be filled with 0’s.

Rpin is 0-15 for an I/O pin, or 16 for the internal serial port (pin 2, RX). If a regular I/O pin is used, then the input will be on a TTL-level pin. If the RX pin is used, then the input will be on a pseudo-RS232 pin. The difference is this:

> On a TTL-level pin (0-15), the signals received are 0 to 5 volts. Serial communication with TTL signals should work well for most applications (certainly if the communicating devices are all TTL-level). *If a TTL pin is used to receive RS232-level signals, then the signal should pass through a 22K resistor before reaching the pin (otherwise, the TTL pin may be damaged).*
>
> On the RS232-level pin (16), the signals received are -10 to 10 volts. Serial communication with RS232 signals is standard on most serial devices, such as printers, modems, etc. This pin is designed to accept RS232 signals, so no in-line resistor is necessary.
>
> In either case, the low/high threshold is at 1.4 volts (anything below reads as 0, anything above reads as 1).

Special note: if the RS232 serial port (TX and RX) is used AND the Stamp II is connected to a PC via the normal serial programming cable AND the PC is running a terminal program, then you may experience odd behavior when trying to use the terminal program to communicate with the Stamp II. This is because many terminal programs hold the DTR line high, which is the same line used to reset the Stamp II prior to programming. If you are experiencing this problem, you should disconnect the DTR line (pin 3 on the Stamp II) when using the terminal program.

Fpin is an optional flow-control pin. This pin may be used to implement "flow-control" handshaking between serial devices.

Baudmode is a composite value which specifies baud rate, parity mode, and true/inverted input.

Bits 0-12 of *baudmode* are the serial bit period expressed in microseconds-20. Bit 13 ($2000) is 0 for 8-bit no parity, 1 for 7-bit parity. Bit 14 ($4000) is 0 for true, 1 for inverted. Bit 15 ($8000) is not used for SERIN.

If the RS232-level pin (16) is specified, then *baudmode* bit 14 is ignored. However, it will still affect operation of the optional flow-control pin *(fpin)*.

An easy formula for calculating the bit period (bits 0-12) is:

$$\text{int}\left(\frac{1{,}000{,}000}{\text{baud rate}}\right) - 20$$

Example values for *baudmode:*

3313	300 baud, 8-bit, no parity, true data
3313 +$2000	300 baud, 7-bit, parity, true data
3313 +$4000	300 baud, 8-bit, no parity, inverted data
3313 +$6000	300 baud, 7-bit, parity, inverted data
813	1200 baud, 8-bit, no parity, true data
396	2400 baud, 8-bit, no parity, true data
188	4800 baud, 8-bit, no parity, true data
84 +$6000	9600 baud, 7-bit, parity, inverted data
32	19200 baud, 8-bit, no parity, true data
6	38400 baud, 8-bit, no parity, true data

Plabel is an optional label that specifies where to branch in the event of a parity error (parity mode must be enabled).

Timeout is an optional value that specifies how long to wait before giving up and branching to *tlabel.* The amount of time is measured in milliseconds, and may be up to 65535 (approximately 65 seconds).

Inputdata follows the conventions below:

variable	Input a byte and store in variable.
STR bytearray\L{\E}	Input a string into bytearray of length L with optional end-character of E (0's will fill remaining bytes).
SKIP L	Input and ignore L bytes.
WAITSTR bytearray	Wait for bytearray string (bytearray is terminated by 0).
WAITSTR bytearray\L	Wait for bytearray string of length L.
WAIT (value,value,...)	Wait for up to a six-byte sequence.
DEC variable	Input decimal value.
DEC1-DEC5 variable	Input decimal value of fixed length.
SDEC variable	Input signed decimal value.
SDEC1-SDEC5 variable	Input signed decimal of fixed length.
HEX variable	Input hex value (i.e., "00F3").
HEX1-HEX4 variable	Input hex value of fixed length.
SHEX variable	Input signed hex value.
SHEX1-SHEX4 variable	Input signed hex value of fixed length.
IHEX variable	Input indicated hex value (i.e., "$00F3").
IHEX1-IHEX4 variable	Input indicated hex value of fixed length.
ISHEX variable	Input signed indicated hex value.
ISHEX1-ISHEX4 variable	Input signed indicated hex value of fixed length.
BIN variable	Input binary value.
BIN1-BIN16 variable	Input binary value of fixed length.

SBIN variable	Input signed binary value.
SBIN1-SBIN16 variable	Input signed binary value of fixed length.
IBIN variable	Input indicated binary value (i.e., “%1101”).
IBIN1-IBIN16 variable	Input indicated binary value of fixed length.
ISBIN variable	Input signed indicated bin value.
ISBIN1-ISBIN16 variable	Input signed indicated bin value of fixed length.

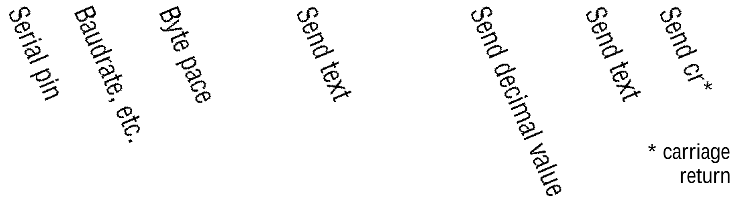
SEROUT tpin,baudmode,{pace,}[outputdata]
SEROUT tpin\fpin,baudmode,{timeout,tlabel,}[outputdata]

Output data serially.

---

example:

```
SEROUT 3,84+$4000,100,[“The temperature is ”,dec temp,“ degrees.”,cr]
```



Send serial data on pin 3. Transmission mode is 9,600 baud, 8 data bits, no parity, inverted data, driven output. Transmitted bytes will be sent at the rate of one byte every 0.1 seconds (100 milliseconds). Sample output would look like this: “The temperature is 75 degrees.”

Tpin is 0-15 for an I/O pin, or 16 for the internal serial port (pin 1, TX). If a regular I/O pin is used, then the output will be on a TTL-level pin. If the TX pin is used, then the output will be on a pseudo-RS232 pin. The difference is this:

> On a TTL-level pin (0-15), the signals sent are 0 to 5 volts. Serial communication with TTL signals should work well for most applications (certainly if the communicating devices are all TTL-level). *However, if the device receiving serial data requires real RS232 voltages, then it may not accept data from a TTL pin on the Stamp II.*
>
> On the RS232-level pin (16), the signals sent are RX to 5 volts. This may seem a bit confusing, but this is how the circuit works: for sending a low signal, the Stamp II reflects the “rest” (low) state of the RX pin, which is usually about -10 volts; for sending a high signal, the Stamp II uses 5 volts. This system only works if the serial device to which the Stamp II is connected has “real” +/- voltage generation circuitry. Real RS232 voltages are standard on most serial devices, such as PCs, printers, modems, etc.

Fpin is an optional flow-control pin. This pin may be used to implement "flow-control" handshaking between serial devices.

Baudmode is a composite value which specifies baud rate, parity mode, true / inverted output, and open / driven output.

Bits 0-12 of *baudmode* are the serial bit period expressed in microseconds-20. Bit 13 ($2000) is 0 for 8-bit no parity, 1 for 7-bit parity. Bit 14 ($4000) is 0 for true, 1 for inverted. Bit 15 ($8000) is 0 for driven, 1 for open-drain / open-source.

If the RS232-level pin (16) is specified, then *baudmode* bits 14 and 15 are ignored. However, their settings will still affect operation of the optional flow-control pin *(fpin)*.

An easy formula for calculating the bit period (bits 0-12) is:

$$\text{int}\left(\frac{1{,}000{,}000}{\text{baud rate}}\right) - 20$$

Example values for *baudmode:*

3313	300 baud, 8-bit, no parity, true data, driven output
3313 +$2000	300 baud, 7-bit, parity, true data, driven output
3313 +$4000	300 baud, 8-bit, no parity, inverted data, driven output
3313 +$6000	300 baud, 7-bit, parity, inverted data, driven output
3313 +$8000	300 baud, 8-bit, no parity, true data, open drain/source
813	1200 baud, 8-bit, no parity, true data, driven output
396	2400 baud, 8-bit, no parity, true data, driven output
188	4800 baud, 8-bit, no parity, true data, driven output
84 +$6000	9600 baud, 7-bit, parity, inverted data, driven output
32	19200 baud, 8-bit, no parity, true data, driven output
32 +$2000	19200 baud, 7-bit, parity, true data, driven output
6	38400 baud, 8-bit, no parity, true data, driven output

Pace is an optional value that specifies how long to wait between transmitted bytes. The amount of time is measured in milliseconds, and may be up to 65535 (approximately 65 seconds). A pace value may only be specified if flow-control is not being used.

Timeout is an optional value that specifies how long to wait before giving up and branching to *tlabel*. The amount of time is measured in milliseconds, and may be up to 65535 (approximately 65 seconds). A timeout value and label are only applicable if flow-control is being used.

Outputdata follows the DEBUG formatting conventions. Please refer to the DEBUG description earlier in this manual.

SHIFTIN

Shift bits in synchronously.

usage: SHIFTIN dpin,cpin,mode,[variable{\bits},...]

Dpin is the data input.

Cpin is the clock output.

Mode is 0 for msb-first / pre-clock, 1 for lsb-first / pre-clock, 2 for msb-first / post-clock, or 3 for lsb-first / post-clock.

Variable receives the shifted-in data.

Bits is an optional bit count. Values from 1-16 can be given; 8 is the default.

The following symbols are defined for use with SHIFTIN:

Symbol	Value
MSBPRE	0
LSBPRE	1
MSBPOST	2
LSBPOST	3

SHIFTOUT

Shift bits out synchronously.

usage: SHIFTOUT dpin,cpin,mode,[data{\bits},...]

Dpin is the data output.

Cpin is the clock output.

Mode is 0 for lsb-first or 1 for msb-first.

Data is a value to be shifted out.

Bits is an optional bit count. Values from 1-16 can be given; 8 is the default.

The following symbols are defined for use with SHIFTOUT:

Symbol	Value
LSBFIRST	0
MSBFIRST	1

Data is shifted out at approximately 16 kbits per second.

SLEEP

Sleep for x seconds (x=0 to 65535).

usage: SLEEP x

Enter low-power mode and keep I/O's updated. Every ~2.3 seconds, the I/O's will go high-z for ~18ms. Approximately 50µa average current will be consumed.

When x seconds have been accrued in SLEEP mode, execution continues at the next instruction. Though the granularity of SLEEP is ~2.3 seconds, the error is within 1% over extended periods of time.

STOP

Stop execution.

usage: STOP

Execution is frozen, but low-power mode is not entered. This is like END, except that the I/O's never go high-z; they remain driven.

Hardware reset will end STOP.

TOGGLE

Make a pin an output and toggle its output state.

usage: TOGGLE pin

Pin is 0-15.

The pin will be made an output and its output state (0 or 1) will be toggled: output of 0 is changed to 1; output 1 is changed to 0.

WRITE

Write a byte into the EEPROM.

usage: WRITE location,byte

Location is 0-2047. *Byte* is 0-255. *Byte* will be written into *location.*

XOUT

Output X-10 powerline control codes to a PL513 or TW523 powerline interface module.

usage: XOUT mpin,zpin,[house\keyorcommand{\cycles,...]

Mpin will be made a low output and is the modulation control.

Zpin will be made an input and is the zero-crossing detect.

House is the house code (0-15 is 'A'-'P').

Keyorcommand is a key (0-15 is '1'-'16') or command (see table below).

Cycles is an optional number which overrides the default of two; this should only be used with 'dim' and 'bright' commands.

X-10 Command (symbol)	Value
UNITON	%10010
UNITOFF	%11010
UNITSOFF	%11100
LIGHTSON	%10100
DIM	%11110
BRIGHT	%10110

The following chart gives the wiring connections for connecting the Stamp II to a PL513 or TW523 powerline interface module.

Powerline Interface Pin	Stamp II Pin
1	Zpin
2	GND
3	GND
4	Mpin

Application note #1 gives a good description of the X-10 system, as well as how to use it with the Stamp II.