Stamp Applications no. 16 (June '96):

Using BS2 Serial Communication: Serin and Serout Demystified

Plus the ABCs of ASCII Characters

by Scott Edwards

EVERYONE LOVES TO HATE instruction manuals. In the Stamp world, users are particularly unhappy with the manual's explanation of the BASIC Stamp II's deluxe versions of Serin and Serout.

Rather than rag on the manual, let's review the fundamentals of serial comms and see if we can crack the code ourselves.

In BASIC-for-Beginners, we'll look at a related subject—the ASCII character set.

General Serial Theory. Serin and Serout are used for *asynchronous* serial communications. In this method, bits travel one at a time over a single wire.

Bits are sent at a precise speed, called the *bit rate* and expressed in bits per second (bps) or *baud*. The reciprocal of the bit rate is the *bit time* or *bit period*—the amount of time allotted to each bit. For instance, at 2400 bps, each bit gets 1/2400 = 416.67 microseconds (µs). This timing must be pretty accurate for a serial link to work properly, since timing is the only thing that distinguishes one bit from the next.

The serial signal has two states, traditionally called *mark* and *space*. In RS-232 serial, mark is a negative voltage (–10V) and space is a positive voltage (+10V).

When no data is being transmitted, the signal line idles in the mark (negative) state, also known as the *stop-bit* condition. To start a transmission, the sender switches to the *start-bit* condition, which is a space (+). It holds the signal line in that state for one bit time.

After the start bit come the data bits. A 1 is represented by a mark (–), and a 0 by a space (+). Each bit is output for exactly one bit time.

The number of data bits can vary. Some older equipment used as few as five data bits; today it's usually eight, but sometimes seven for a text-only setup.

After the data bits, there can be a *parity* bit, a means of error checking. The transmitting computer counts the 1s in the data bits, and sets or clears the parity bit to make that count total an even number. That's called, appropriately, *even* parity. The receiver can perform the same count, and see if it jibes with the received parity bit. If not, the data (or the parity bit) was received incorrectly.

There are several other parity schemes: *odd* (opposite of even), *space* or *zero* parity (parity bit is always 0), *mark* or *forced* parity (parity bit is always 1), or no parity (parity bit left out entirely).

The last bit of a serial transmission is the *stop bit*. It's a return to the mark condition that existed before the start bit. Even if the sender wants to transmit more data right away, it must wait one bit time in the stop-bit condition.

Note that some serial devices allow you to specify more than one stop bit. Settings of 1, 1.5 and 2 or more are pretty common. All this means is that the signal will remain in the stop-bit condition for more than one bit time. This additional breathing room between data transmissions may be required to give the receiver extra processing time.

The use of opposite start and stop bits is the key to asynchronous communication. The receiver must identify each incoming bit by its precise time of arrival. It bases its reckoning of time on the exact instant of the signal's transition from the stop- to start-bit state. This allows it to reset its timing with each new arrival of data (called a *frame*) so that small timing errors can't accumulate over many frames to become big timing errors.

Figure 1 shows a frame of serial data—8 data bits, no parity—in oscilloscope fashion. The data sent is 01101001 binary—the code for letter "i".

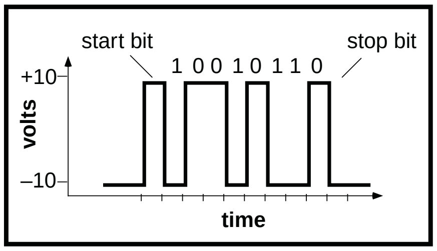

Figure 1. One frame of serial data.

Not every device that can receive serial data can process it as fast as it might arrive. And some devices, like the Stamps, must devote all of their attention the serial line in order to receive data at all. In such cases, the devices may use one or more *handshaking* bits to indicate whether or not they are ready to receive. Serial senders are expected to obey these signals, and send data only when the receiver is ready. Since computer-science folks like to call serial inputs and outputs *streams* of data, another term for handshaking is *flow control*.

BS2 Serial Theory, Serout. The BS2 has a somewhat bewildering array of serial options. Let's apply our knowledge of serial comms to understanding them.

The absolute minimum you must specify in order to send a byte of data from the BS2 are the pin through which to transmit; the baud rate and mode, combined into a single value that Parallax calls the *baudmode*; and the data to send.

If you're familiar with other computers but not the Stamps, you may be a little puzzled by the need to specify the pin. Does this mean that the Stamp has multiple serial ports? No. The Stamps do away with the whole concept of dedicated serial ports, instead treating serial I/O as just another function of the normal I/O pins. The BS2 does, however, regard its serial programming connector as a special case—you can use it for serial comms, but not for conventional pin I/O.

Baudmode is a 16-bit numbers that specifies all of the important characteristics of the serial transmission: the bit time, data and parity bits, polarity, and drive characteristics. I'll explain how this works, but if you want to skip the gory details, I'll understand. Just look up the combination of standard baud rates and characteristics in table 1.

As shown in figure 2 below, the lower 13 bits of the baudmode specify the bit period in microseconds. The BS2's internal process for organizing and sending the bits apparently involves 20µs of overhead, so the actual bit period is the baudmode-specified value, plus 20. As the manual points out, you can calculate the bit-period part of the baudmode as 1,000,000/bit rate, minus 20. Use only the integer portion of the result; drop any numbers to the right of the decimal point.

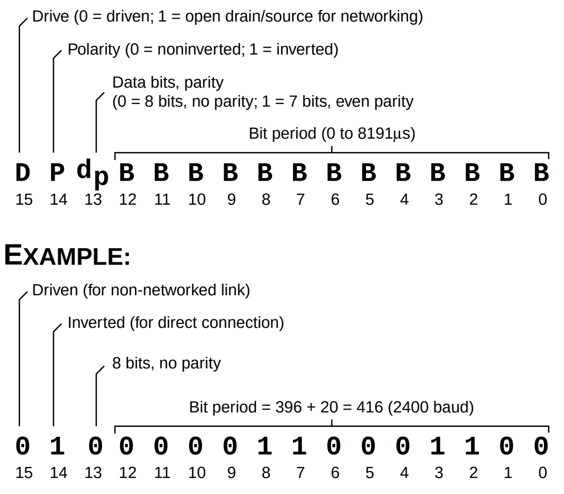
Drive (0 = driven; 1 = open drain/source for networking)
Polarity (0 = noninverted; 1 = inverted)
Data bits, parity
(0 = 8 bits, no parity; 1 = 7 bits, even parity
Bit period (0 to 8191µs)
D P d p B B B B B B B B B B B B B
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
EXAMPLE:
Driven (for non-networked link)
Inverted (for direct connection)
8 bits, no parity
Bit period = 396 + 20 = 416 (2400 baud)
0 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure 2. BS2 Baudmode dissected.

Table 1. Baudmodes for Standard Serial Rates and Modes

	Baudmodes (decimal numbers)			
	Direct Connection		Through RS-232 Line Driver	
Baud Rate	8 data bits, no parity	7 data bits, even parity	8 data bits, no parity	7 data bits, even parity
300	19697	27889	3313	11505
600	18030	26222	1646	9838
1200	17197	25389	813	9005
2400	16780	24972	396	8588
4800	16572	24764	188	8380
9600	16468	24660	84	8276
19200	16416	24608	32	8224
38400	16390	24582	6	8198

The three other bits of the baudmode serve as switches for the following serial options:

Data bits and parity. The BS2 supports the two most popular combinations of data bits and parity: 8 data bits, no parity (abbreviated 8N); and 7 data bits, even parity (7E). A 0 in bit 13 of the baudmode selects 8N; a 1 selects 7E. An easy way to set this bit is by adding or logically ORing the hex value $2000 with the bit period. (In Stamp notation, hex numbers begin with the dollar sign $).

Polarity. In our discussion of serial theory above, you may have noticed that the voltages were backward, with –10V meaning 1 and +10V meaning 0. RS-232 signals are inverted with respect to the digital convention of 0V = 0 and 5V = 1. So when you want to talk directly to an RS-232 device, you must select inverted polarity by putting a 1 in bit 14 of the baudmode (by adding or ORing hex $4000 into it). RS-232 line drivers, chips designed for converting digital signals to RS-232 voltages, also invert those signals. So to talk through a line driver you'd put a 0 in bit 14.

Note that if you select the built-in serial connector (pin 16) as the destination of Serout, the polarity bit has no effect on the output. It's always sent inverted, which is correct for a direct connection.

Drive. So far in our discussion of serial communication, we've assumed that there were just two devices involved, one talker and one listener. We haven't discussed what might happen if the listener interrupted the talker (by outputting data serially onto the same wire at the same time), or if there were multiple talkers. Those are the kinds of situations that arise when you set out to construct a network.

While networking is beyond the scope of this article, you should know that the BS2's serial commands can be configured to support it. Writing a 1 to bit 15 of the baudmode (by adding or ORing hex $8000 into it) selects open-drain or open-source signaling, depending on the polarity. The important thing to know about these "open" settings is that they only drive to one power-supply rail or the other, but not both. This can be used to let multiple Stamp drive the same signal line without damaging each other. See Parallax application note no. 14 for more on networking illustrated with an example for the BS1.

All of the examples we'll present here will be non-network situations in which the serial pin is always driven to the appropriate power-supply rail.

Serout Examples. Armed with all of the foregoing information, we're ready to craft some BS2 Serouts. The general form of the instruction is:

```
Serout tpin,baudmode,{pace,}[outputdata]
```

where tpin is the pin to transmit through; baudmode is the setting for bit rate, data bits/parity, polarity and drive; pace is an optional delay in milliseconds (ms) between bytes; and output data is, well, output data. For example:

```
Serout 1,16468,["HELLO!"]
```

would transmit the message HELLO! through pin 1 at 9600 baud, 8N, inverted, with non-network drive. (Baudmode from table 1.) Another example:

```
Serout 1,16468,[65]
```

This time, the output data is the value 65. If you were looking at the serial output of this instruction on a terminal (serial communication program) screen, what would you see? If you said “65” make an obnoxious buzzer sound, ’cause you’re wrong. Serout would send a single byte whose bits total up to equal the decimal number 65. A terminal screen displays this byte by looking up the corresponding symbol from the ASCII character set, which is “A.” If you really wanted to see “65” instead, you would write:

```
Serout 1,16468,[DEC 65]
```

The DEC modifier, short for “decimal” tells PBASIC to convert the single-byte value 65 into text representing the decimal number 65. There are 24 of these nifty formatting modifiers listed under Debug in the BS2 manual. (Debug is just a special case of Serout.) Ready for another?

```
numba   var     byte
numba = 65
Serout 1,16468,[numba,cr]
Serout 1,16468,[DEC numba]
```

The first Serout sends the contents of the byte variable numba. Since numba contains 65, a terminal would show “A.” The symbol “cr” after numba represents the value 13, which is the ASCII code for a carriage return. So the receiving terminal would return to the left margin of the screen. If set up properly, it would also hop down to the next line, but that issue is between you and the maker of your terminal software.

The second Serout would cause the text “65” to appear on the screen.

Let’s return to our original example for a moment. Suppose you wanted to drag out the message, sending one character per second. You’d just need to add a optional pacing value, like so:

```
Serout 1,16468,1000,["HELLO!"]
```

The BS2 would insert a 1000-ms delay after every byte: H <pause> E <pause>... Why would you want to do this? Probably to accommodate a device that needed time to digest each byte after it arrived. For example, say the BS2 was talking to a BS1, which took each incoming byte and copied it to EEPROM. The BS1 would need more than one stop-bit time to prepare for the next byte, so pacing could save the day.

If you want to communicate between BS2s, there’s an even better option: flow control. With flow control, the receiving BS2 tells the sending BS2 when to transmit each byte. The general form of a flow-controlled Serout is:

```
Serout tpin\fpin,baudmode,{tOpt,}[outputdata]
```

where fpin is the pin that controls the flow of serial-output data. We’ll talk about the timeout options—tOpt in the example above—later. Fpin’s control logic depends on the data polarity established by baudmode; if it’s set for noninverted data, then a 0 on fpin starts data transmission and 1 stops it. If polarity is set for inverted data, a 1 on fpin starts transmission and 0 stops it. Note that the polarity sets the logic of the fpin even if Serout’s destination is the built-in serial connector (which always outputs inverted for compatibility with PC serial ports).

Suppose you connected two BS2s together for flow-controlled serial communications—pins P0 to P0 and P1 to P1. The programs’ Serout and Serin instructions might look like this:

TRANSMITTER:

```
Serout 0\1,16468,["HELLO!"]
```

RECEIVER :

```
Serin 0\1,16468,[variableName]
```

On the receiving end, we’re assuming that the Serin instruction is enclosed within a loop which will repeat to gather up each byte of the transmitted message. Otherwise, it would grab just the first byte, and leave the transmitter in the lurch waiting for permission to send the rest.

That possibility, that the receiver would stop asking for bytes of data before the transmitter has sent its entire message, is the purpose of the timeout options, abbreviated tOpt above. These options specify two things: how long the BS2 should wait for permission to send a pending byte of data, and what part of the program it should go to in the event that permission doesn't arrive in time.

Here's a Serout example with the timeout options enabled.

```
Serout 0\1,16468,100,xmitFailed,["HELLO!"]
```

The portions in boldface type are the timeout options. The first specifies that the BS2 will wait 100 milliseconds; the second tells it to go to the program label xmitFailed in the event that more than 100 ms passes without permission to send one of the bytes of the message.

A couple of final notes on flow-controlled Serouts are in order. First, beware of using the wrong slant bar between tpin and fpin. You want a backslash (\) not a forward slash(/). PBASIC won't generate an error if you use a forward slash, because it will think that you're specifying the tpin number as one value divided by another: x/y.

Second, be aware that you cannot combine flow-control with pacing. The reason should be obvious—if the state of a pin is telling the BS2 when to send the next byte, a pacing delay is irrelevant.

BS2 Serial Theory, Serin. Once you understand the Serout options, Serin is pretty easy. The basic form is:

```
Serin rpin,baudmode,[inputdata]
```

where rpin is the pin to receive through; baudmode is the setting for bit rate, data bits/parity, polarity and drive; and input data specifies what to do with the incoming data (ignore it; compare it to a string; store it in a variable; or convert it from text to a numeric value and then store it in a variable). Let's look at a simple example:

```
serData    var    byte
Serin 1,16468,[serData]
```

The baudmode—the same one used in the examples above—sets 9600 baud, 8N, inverted. Serin would wait for and receive a single byte of data through pin 1 and store it in the byte variable serData. Here's a variation:

```
serData    var    byte
Serin 1,16468,[DEC serData]
```

In this case, the DEC modifier tells Serin to wait for numeric text (like typing the numbers "1,2,3..." from a terminal keyboard) and to convert that text into a one-byte value (range 0 to 255). There are quite a few input formatting modifiers for decimal, hex, and binary numbers. See the manual for a complete list of choices.

In the theory discussion above, we talked about parity serving as a simple means of error checking. If you specify a 7E baudmode, the BS2 will accept the parity bit. Optionally, you can tell the BS2 what part of your program to go to in the event of a parity error. Just put the label for the routine right after baudmode, like so:

```
Serin 1,24660,parityErr,[inputdata]
```

If the data is received correctly—or at least the parity bit *indicates* that it was received correctly—your program will continue on the next line. If the parity bit is incorrect, the program will go to the label parityErr.

If you're using a non-parity baudmode, Serin will not go to parityErr. And if you do specify parity, but don't supply a label to go to in the event of an error, PBASIC ignores the error.

Serin includes a flow-control option that complements the one supported by Serout. As we showed in the Serout section above, you can connect one BS2 to another and communicate with byte-by-byte handshaking. For example, suppose the program surrounding the "Receiver" code in the Serout section looked like this:

```
letta      var       byte
again:
  Serin 0\1,16468,[letta]
  debug letta
  pause 1000
goto again
```

The two BS2s would cooperate perfectly; the letters of the transmitted string would be displayed one at a time at 1-second intervals. That's because the BS2 can turn its serial flow on and off in one-byte doses. So what? Well, PCs generally do not control their serial flow so tightly. They can continue to emit as many as eight bytes of serial data after being told to stop! And some programs ignore hardware flow control entirely. To get finer control over the comm port requires low-level PC programming that's beyond the scope of this article. Just be aware that a BS2 application that depends on byte-by-byte handshaking with a PC is going to be considerably more involved on the PC end than you might expect.

Just as Serout has options for dealing with timeouts, Serin can time out if it doesn't receive data within a prescribed time. Unlike Serout, Serin's timeout option is not tied to flow control. All that matters is whether or not serial data arrived in time. Here's the general form of the instruction:

```
Serin rpin,baudmode,{tOpt,}[inputdata]
```

The timeout option, tOpt, consists of a number of milliseconds to wait, and a program label to go to in the event that no data arrives within that time. For instance:

```
Serin 4,16780,10000,noSerialData,[inputdata]
```

That instruction would accept inverted serial data through pin 4 at 2400 baud. If no data arrived within 10 seconds (10,000 milliseconds), the program would go to the label noSerialData. Otherwise, it would receive the data and continue with the next instruction.

Serin's timeouts can be combined with any of its other options.

Don't be cookbook-codependent! The BS2's Serin and Serout instructions have so many possible variations and combinations that I almost certainly didn't show you exactly what you need for your next application. And I could fill every page of this magazine, and still not hit every possibility. Remember that there are 65,536 possible baudmodes alone!

My point is that you shouldn't depend on explicit recipes for programming. A nudge in the right direction should be enough. Don't be afraid to experiment, explore blind alleys, make mistakes, or even waste some time getting your project working. Don't condemn the time you spend figuring things out on your own as "reinventing the wheel." What you're really doing is learning in a way that carves deep gullies in your gray matter.

This pep talk arises out of my concern over things I see on the Internet and communications I receive from readers. Many people seem to be so terrified of doing original work that they spend more time searching for someone else's solution than they would formulating their own. They may accomplish the task at hand with a borrowed answer, but they deprive themselves of the opportunity to improve their skills.

BASIC for Beginners. Last month I took you on a whirlwind tour of the decimal, binary, and hexadecimal numbering systems. You saw that there's more than one way to represent a given number.

This time we'll look at how numbers themselves can represent text, like ABC, *@%^! and even 1,2,3...

Digital circuits, including Stamps, can only manipulate bits. When we looked at systems of numbers, we could readily see how bits could be clumped together into groups like bytes and words and used to express numbers. A byte, consisting of eight bits, can express 256 different values from 0 to 255.

But one of the most common uses of desktop computers is to process text. How do they handle that?

As far as the processor is concerned, there is no distinction between text and numbers. Everything consists of bits, bytes, and words (clusters of 16 or more bits). It's the job of a program to decide whether a particular piece of data is a number or a chunk of text.

Since text-based applications are so important, the computer industry early on worked out a standard system of numbers to represent the keys of a typewriter keyboard as well as special control characters significant to devices like printers and display terminals.

Table 2. The ASCII Character Set

Control Codes			Printing Characters					
Name/Function	*Char	Code	Char	Code	Char	Code	Char	Code
null	NUL	0	<space>	32	@	64	‘	96
start of heading	SOH	1	!	33	A	65	a	97
start of text	STX	2	"	34	B	66	b	98
end of text	ETX	3	#	35	C	67	c	99
end of xmit	EOT	4	$	36	D	68	d	100
enquiry	ENQ	5	%	37	E	69	e	101
acknowledge	ACK	6	&	38	F	70	f	102
bell	BEL	7	’	39	G	71	g	103
backspace	BS	8	(	40	H	72	h	104
horizontal tab	HT	9	)	41	I	73	i	105
line feed	LF	10	*	42	J	74	j	106
vertical tab	VT	11	+	43	K	75	k	107
form feed	FF	12	,	44	L	76	l	108
carriage return	CR	13	-	45	M	77	m	109
shift out	SO	14	.	46	N	78	n	110
shift in	SI	15	/	47	O	79	o	111
data line escape	DLE	16	0	48	P	80	p	112
device control 1	DC1	17	1	49	Q	81	q	113
device control 2	DC2	18	2	50	R	82	r	114
device control 3	DC3	19	3	51	S	83	s	115
device control 4	DC4	20	4	52	T	84	t	116
negative acknowledge	NAK	21	5	53	U	85	u	117
synchronous idle	SYN	22	6	54	V	86	v	118
end of xmit block	ETB	23	7	55	W	87	w	119
cancel	CAN	24	8	56	X	88	x	120
end of medium	EM	25	9	57	Y	89	y	121
substitute	SUB	26	:	58	Z	90	z	123
escape	ESC	27	;	59	[	91	{	124
file separator	FS	28	<	60	\	92	|	125
group separator	GS	29	=	61	]	93	}	126
record separator	RS	30	>	62	^	94	~	127
unit separator	US	31	?	63	_	95	<delete>	128

* Note that the control codes have no standardized screen symbols. The characters listed for them are just names used in referring to these codes. For example, to move the cursor to the beginning of the next line of a printer or terminal often requires sending linefeed and carriage return codes. This common pair is referred to as “LF/CR.”

This system is called the American Standard Code for Information Interchange, ASCII (asskey) for short. ASCII traditionally uses seven bits of a byte to provide a total of 128 different codes, but most computers also recognize and use the additional 128 possible byte codes to extend the ASCII symbol set with graphics symbols, foreign-language characters, and special punctuation. These so-called *high-ASCII* characters are not standardized.

Table 2 presents the ASCII character set, with its code-number equivalents.

Knowing about ASCII becomes important when you are using the Stamp for serial communication. If you want to send the value of a byte variable, you must decide whether you want it sent as text—a sequence of ASCII symbols—or as a single byte.

An example: Suppose byte variable b2 contains 105. You use Serout to send it to your PC running a terminal communication program. On a BS1-type stamp, the instruction would be:

```
Serout 0,N2400,(b2)
```

The Stamp transmits eight bits: 01101001. Your PC displays a single character: “i.” The terminal program is expecting text, so it interprets that byte as the its ASCII symbol (table 2).

If you wanted the PC to display a human-readable number showing the value of b2, you would change that instruction to read:

```
Serout 0,N2400,(#b2)
```

The number sign (#) tells PBASIC to convert the binary value of b2 into corresponding decimal digits, and to send the ASCII symbols for those digits out serially. So the Stamp would send three bytes, 00110001 00110000 00110101, to the PC. Those bytes are ASCII codes for 49 48 53, which correspond to the symbols “105.”

Spend a few minutes browsing the ASCII table and notice a few details:

• The numbers 0 through 9 are arranged in order starting at code 48. So to convert a single text digit to its numeric equivalent, all you need to do is subtract 48 from the ASCII code.

• The letters of the alphabet are arranged in order, starting at code 65 for uppercase and 97 for lowercase. If you need to alphabetize text, you simply arrange items in order of their ASCII codes.

• The ASCII codes for corresponding upper- and lower-case letters differ by 32: capital H is code 72, while small h is 72 + 32 = 104. This makes for an easy way to convert lowercase to uppercase; just perform a logical AND of the value with the hex value $0DF. This can be very useful for accepting typed instructions from a terminal program.

There’s no need to memorize the ASCII symbol set; just bear in mind that it exists. And when your Stamp program cusses at you—*@%^!—it may just be reminding you to use #.

NOTE: This article was originally published in 1996. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.