Rubberbands and BAILING WIRE

by Jack Buffington

Getting Keyed Up

How to Use a Standard PC Keyboard to Control Your Robot

Have you ever wanted to send complex commands to your robot without having a computer attached to it? This column will show you how to interface with a standard keyboard that has a PS2 or AT connector at the end of its cable.

Using a PC's keyboard in conjunction with an LCD screen can allow you to easily control the values of variables in your robot or to otherwise modify its behavior. Alternately, if your robot needs dozens of buttons to control it, a PC keyboard will certainly be a cheaper and easier alternative to using standard buttons.

In an ideal world, when you pushed a key on your keyboard you would receive an ASCII character code that corresponded to that key. Unfortunately, there appears to be no obvious rhyme or reason to the codes that you receive when you press a key.

Most keys follow the rule that one byte is sent when they are pressed and, if they are held down for a while, they repeatedly send that same byte. When they are released, they send two bytes. The first is a byte that signifies that a key has been released and the second is the byte that corresponds to that key. That is fairly simple and luckily, most keys follow this rule.

For the most part, the main keyboard and the number pad on the right follow this rule but the arrow keys and keys such as page up or delete don't. Those keys send two bytes when they are pressed, send packets of two bytes if they are held down, and send three bytes when they are released. Things get even more complicated for the 'Pause Break' and 'Print Screen' buttons. The 'Print Screen' key sends four bytes when it is pressed and the 'Pause Break' button sends eight bytes.

Keyboards have sent the same data since the original PC was released, so it is likely that these oddball codes that the keyboard is sending have something to do with making it easier for those computers to process their data. Of course, these days, they make it harder for us to process their data but even still, it is relatively painless to interpret the data that keyboards send.

Let's start by looking at the physical interface between the keyboard and your processor. There are two types of keyboards that you can use: an AT keyboard and a PS2 keyboard. Both of them have the same data format. The only difference between them is the connector at the end of their cable.

There are four wires that must be connected to your project. Two of them are +5 volts and ground. The other two are clock and data. Figure 1 shows the pinouts for these two connectors. The clock and data lines are open collector I/O lines.

You might think of a PC keyboard as being an output only device, but in reality, it can also receive data from the computer. This column won't cover how to send data to a keyboard, but some things that you can tell it to do is to light up its three status LEDs, change its key repeat rate, and request that it resend the last byte that it sent.

Eleven bits are sent for each byte of data that the keyboard transmits. The signals are similar to the signals sent by SPI devices. SPI devices usually play nice and send extra clock cycles if they have a number of bits that don't fit neatly into a multiple of eight. The PC keyboard doesn't do this. It transmits 11 bits and then goes quiet. This means that you won't be able to use a standard SPI peripheral to receive its data.

Before continuing on, let's look at

Figure 1. The PS/2 and five pin DIN keyboard connectors.

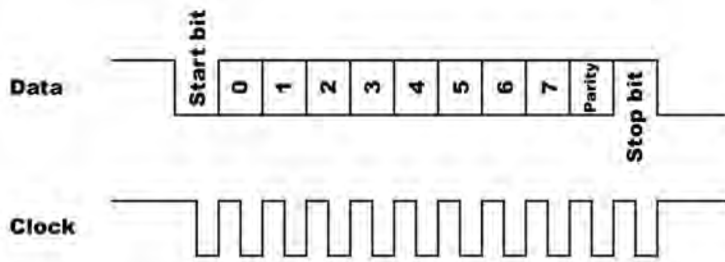


Figure 2. A byte of data being sent from the keyboard.

Byte	Even parity	Odd parity
01010110	0	1
11001011	1	0
00000001	1	0
00000000	0	1

Figure 3. Examples of parity.

the signals that the keyboard is sending. Figure 2 shows a single byte being sent from the keyboard to the host computer.

Some documentation says the PC keyboards are supposed to send their data with a clock that is somewhere in the range of 20 kHz to 30 kHz. The keyboard that was used for this column fell outside of that spec. Its clock rate was 13 kHz. The key point to take from this is that you can't rely on the keyboard to put out its data at a fixed baud rate. The keyboard's clock idles in a high state. Data is considered to be valid on the falling edge of the clock.

The data coming from the keyboard is as follows. The first bit is a start bit. This bit is always low. It is followed by eight bits, which are the actual data. This data is sent with the least significant bit first. The next bit is a parity bit. Specifically, it is an odd parity bit. Parity is a simple way of helping you to verify that you received your data correctly. It can allow you to detect single bit errors. If you are using odd parity, the sum of the 1s in the byte plus the parity bit will be an odd number. With even parity, the sum of the 1s in the byte plus the parity bit will be an even number.

Getting back to how to receive data, look at the chunk of code in Figure 4 that shows one way to receive keyboard commands. This code is pretty wasteful of your processor, though, since it hogs 100% of it. Still, it lays the groundwork for the next method that will be shown. In neither version is the parity bit used. If you are experiencing problems with the data that you are receiving, you might want to put in a check to make sure that the parity is indeed correct.

The code in Figure 4 works, but there is a better way to collect this data. Since the keyboard is supplying the clock pulses, we can connect the clock line to an I/O pin on your processor that can generate an interrupt when that pin changes. This particular piece of code uses the PIC's port B interrupt which happens whenever a pin that is set to be an input on port B pins 4 through 7 changes state. Since we only want to look at the data when the clock line has fallen, the interrupt routine simply returns when the clock is high.

The way that this interrupt routine works is that it forms a simple state machine that keeps track of which bit is being received at any given time. Each time that the interrupt happens, if the clock is falling, then it uses a switch statement to jump to code that does what is appropriate for that particular bit. It

TECH TIDBIT

If an output is open collector, then it will only be able to pull the signal line low so you will need to have a 'pull-up' resistor to make that output actually be able to send data. A pull-up resistor is not a special type of resistor. It is simply any resistor that is connected between the positive supply (usually five volts) and the output; 10K resistors work well as pull-up resistors.

If you go higher in value, you may experience noise. If you go lower, your circuit will draw more power when the output is being driven low. While at first open collector outputs might seem like the manufacturer was just being lazy when implementing their hardware, it actually allows their device to be connected to devices that have a different operating voltage without any problems.

Figure 4. Code that receives keyboard data.

```
int8 I, theByte;
int16 RXdata;

For(I = 0; I < 11; I++)
   {
   while(input(KEYCLOCK)) {}     // do nothing while the clock is high

   if(input(KEYDATA))
      bit_set(RXdata,10);
   else
      bit_clear(RXdata,10);
   rxData /=2

   while(!input(KEYCLOCK)) {}    // do nothing for the remainder of the time that
                                 //the clock is low
   } // end of for loop

// the result is now in the low byte of RXdata.
theByte = *(&RXdata);    // grabs the low byte
theByte = RXdata;        // another way to grab the low byte.  Since the upper bits
                         // don't fit into an 8-bit variable, they are simply truncated.
```

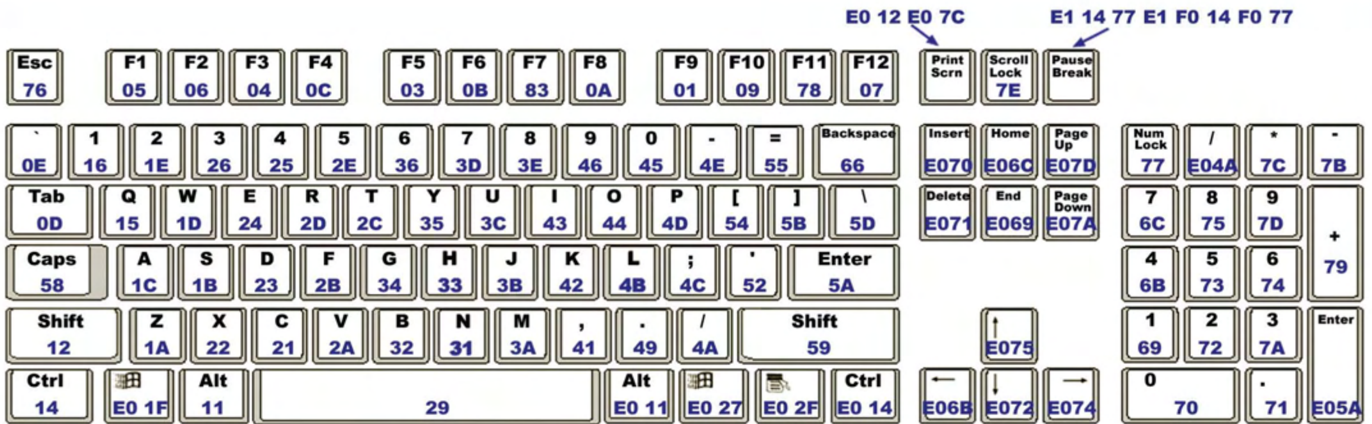


Figure 5. The codes returned by each key on the keyboard.

keeps track of which bit is being operated on using a global variable. Since 8 of the 11 clock cycles load a bit into the RXbyte variable, a function was created to load the bit so that the code didn't need to be repeated eight times.

Once the byte is loaded, it is put into a ring buffer so that the main routine can process the byte when it is ready. You can download a copy of the code for this column from *SERVO*'s website (**www.servomagazine.com**) to learn more about the ring buffer. Ring buffers were covered in a previous column so that code won't be discussed here.

In a perfect world, once a byte was received you could go ahead and process it as an ASCII character, but that is not the case so you will need to do a little work on the data that you receive from the keyboard before using it. Each button on the keyboard puts out a specific character code. You might assume that it would put out different codes for capital and lower case letters, but you would be wrong. You will need to pay attention to the shift and caps lock keys yourself in order to be able to use lower and upper case letters, as well as characters such as #, $, and % that are located on keys with multiple symbols.

To convert the keys into ASCII codes, you will need to run the bytes that the keyboard sends through one of three lookup tables to figure out what the ASCII code would be. The first lookup table is for when neither the shift is pressed and the caps lock key is not pressed. This table returns the lower case letters and the numbers. The second lookup table contains ASCII codes for upper case letters and the symbols on the tops of the keys with two symbols. The final lookup table is for if the shift key is not being pressed but caps lock is on. This table has capital letters and the numbers in it.

These tables will allow you to print out characters as a typewriter would. If you want to replicate how a PC works where if caps lock is on and you push the shift key then you get lower case

TECH TIDBIT

Interrupts allow your processor to do multiple things without too much difficulty. When an interrupt happens, the processor will stop what it was doing and jump to an interrupt routine. Some processors will jump to a specific section of code for each interrupt and other processors will jump to one general-purpose interrupt location where it will be up to you to figure out what generated the interrupt. Interrupts are usually triggered by the processor's peripherals, such as a serial port receiving data or an analog-to-digital conversion finishing. When the processor is done running the interrupt code, it will return to where it was before the interrupt happened.

"...so WAKE UP! READ THIS! It's IMPORTANT!"*
"If these resistors weren't in place, your Photopopper would simply give up and then start daydreaming about a vacation in the tropics."*
"So please follow these next steps with due care and diligence, or I'll have to sick my pet frog "Guido" on you..."*
"Bad Sumovore builder, BAD!"*
"Use a pen, pencil, pricked finger, chocolate bar - anything to mark off the items"*
"Everybody wants to start with the most important looking parts. Well, tough - you can't."*
"There! Go forth and terrorize the world with mutant modified servos!"*
"Start by using a sharp knife or razor to cut a small chunk of the chocolate bar you've got hidden in your desk, and eat it."*
"Igor! Bring me a brain! And a pair of eyes while you're at it..."*
"There are bite marks on my Sumovore!"*
Warped Humour = Fun Kits

SOLARBOTICS.COM™
*Direct quotes from Solarbotics Documentation

letters, then you will need a fourth lookup table to deal with that. The code on *SERVO*'s website only deals with the shift key and ignores the caps lock.

As was mentioned earlier, for most keys you get one byte when a key is pressed and two bytes when a key is released. Let's use the 'A' key as our example. It sends the hexadecimal value 1C when it is pressed. When it is released, it sends two bytes F0 and 1C. The F0 signifies that the key was released. This is the way that all of the one-byte keys work.

Now, let's look at the 'Insert' key. This key sends the two bytes E0 70 when it is pressed. The E0 specifies that it is an extended key. When this key is released, it will send the three bytes E0 F0 70. It first sends the byte E0, which signifies an extended key, then it sends an F0 which indicates that a key has been released, then finally it sends the key code for Insert.

The following two keys are a lot of trouble, but if you are determined to use them, the Print Screen key puts out E0 12 E0 7C when it is pushed and will repeat that code if it is held down. When you release it, it puts out E0 F0 7C E0 F0 12. The Pause Break key is easier. It puts out E1 14 77 E1 F0 14 F0 77 only when it is pressed. It does not repeat and does not put out a code when it is released.

Okay! You now know how to read a PC's keyboard, but what can you do with it? One thing that would be really handy is if you could examine and change the values of some of the variables in your robot without having to drag your computer with you as you followed your robot. You could simply mount an LCD screen somewhere on the robot and include a keyboard connector. This is what the sample code on the *SERVO* website does.

It has an LCD screen and the keyboard connected to a PIC processor. When you type, the appropriate letter or character appears on the screen. You can connect and disconnect the keyboard as often as you want because the pull-up resistors that are on your circuit board will keep the input lines from floating.

One minor thing to keep in mind though is that each time the keyboard is attached to your robot and receives power, it will go through a power on self test and return the value of 170 (AA) to signify that it has passed. You will just need to add

Figure 6. Better code to receive the keyboard's data.

```
#int_RB
RB_isr()
{
 if(!input(KEYCLOCK))
      {// this was a falling transition so go ahead and record the bit if it is a
       // data bit

       switch (whichBit)
          {
          case 0:  // this is the start bit.
             parity = 0;
             if(input(KEYDATA))
                whichBit = 0;   // this wasn't a start bit if KEYDATA was high
             break;             // this will give the code another chance to get it right

          case 1:  // these are the actual data
          case 2:
          case 3:
          case 4:
          case 5:
          case 6:
          case 7:
          case 8:
             loadBit();
             break;
          case 9:  // this is the parity bit
             break;
          case 10: // this is the stop bit.
             if(!isBufferFull())
                putInBuffer(RXbyte);
             break;
          }// end of the switch statement

      if(++whichBit == 11)
          {
          whichBit = 0;
          set_timer0(0);
          }
      }// end of if this was a falling edge
      //set_timer0(0);

}

//————————————————————————————————————————————————————————————————————————————————————————
void loadBit()
   {
   RXbyte /= 2;
   if(input(KEYDATA))
      {
      RXbyte += 128;
      parity++;
      }
   }
```

code in your robot that will ignore this value.

Having a large array of keys in a ready-made package can free you from the tedious task of wiring buttons and figuring out how to connect all of them to your microcontroller in a way that doesn't use all of its I/O pins. In its simplest form, you could use a keyboard so that each key triggered a certain event to happen, such as a light coming on or a motor energizing to drive your robot forward.

Getting more complex, if you wanted to change the values of the variables in your robot, you could just decide on a strategy where, for example, the Q key might increase a variable by one and the A key would decrease it by one.

```
// These are ASCII codes if the shift key is not pressed
const int8 noShift[] =
            {  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  9, 96,  0,
               0,  0,  0,  0,  0,113, 49,  0,  0,  0,122,115, 97,119, 50,  0,
               0, 99,120,100,101, 52, 51,  0,  0, 32,118,102,116,114, 53,  0,
               0,110, 98,104,103,121, 54,  0,  0,  0, 97,106,117, 55, 56,  0,
               0, 44,107,105,111, 48, 57,  0,  0, 46, 47,108, 59,112, 45,  0,
               0,  0, 39,  0, 91, 61,  0,  0,  0,  0, 15, 93,  0, 92,  0,  0,
               0,  0,  0,  0,  0,  0,  8,  0,  0, 49,  0, 52, 55,  0,  0,  0,
              48, 46, 50, 53, 54, 56, 27,  0,  0, 43, 51, 45, 42, 57,  0,  0};

// these are ASCII codes if the shift key is pressed
const int8 shift[] =
            {  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  9,126,  0,
               0,  0,  0,  0,  0, 81, 33,  0,  0,  0, 90, 83, 65, 87, 64,  0,
               0, 67, 88, 68, 69, 36, 35,  0,  0, 32, 86, 70, 84, 82, 37,  0,
               0, 78, 66, 72, 71, 89, 94,  0,  0,  0, 77, 74, 85, 38, 42,  0,
               0, 60, 75, 73, 79, 41, 40,  0,  0, 62, 63, 76, 58, 80, 95,  0,
               0,  0, 34,  0,123, 43,  0,  0,  0,  0, 15,125,  0,124,  0,  0,
               0,  0,  0,  0,  0,  0,  8,  0,  0, 49,  0, 52, 55,  0,  0,  0,
              48, 46, 50, 53, 54, 56, 27,  0,  0, 43, 51, 45, 42, 57,  0,  0};
```

Figure 7. Lookup tables to convert the codes returned by the keyboard into ASCII codes.

Of course, if you were really ambitious, you might decide to write a command interpreter for your processor so that you could give it commands such as 'set maxspeed = 57'. That might be a bit of overkill, though.

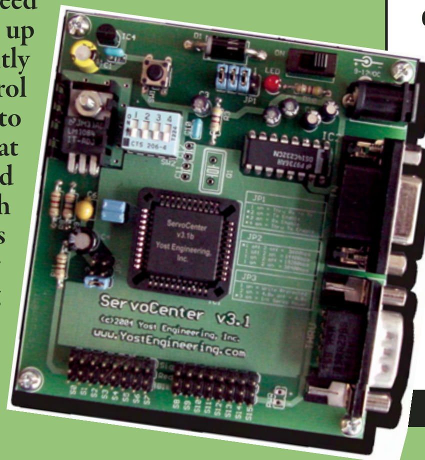
Having a keyboard port on your robot may be a solution to input problems that you are having that won't add much cost or code space to your project. Stay tuned for next month when this column will go over how to use the new Nordic Semiconductor chip that allows for wireless data transfers of up to 1 Megabit per second! **SV**

Taking over the world, one step at a time.

YEI ServoCenter™ 3.1 Controller

USB, Serial, MIDI, or Chip

The ServoCenter™ 3.1 is an embedded controller allowing any device with a serial, USB, or MIDI port to control the seek position and speed of up to sixteen servos per board, up to 256 total servos -- all independently and simultaneously. This control scheme allows each servo to move to its own position, at its own speed, at its own schedule. This unparalleled independent control of both servo position and speed makes ServoCenter 3.1 especially useful for servo control applications including robotics, animatronics, motion control, automation, retail displays, and other areas where independent and coordinated fluid servo motion is desired. Also available in DIP, PLCC, or TQFP packages.



Independent and simultaneous control of the speed and position of 16 RC servos per board, up to 256 total servos

$48.95 - $89.95

Full Package Includes: Power adapter, cable, sample CD, and programming guide.

Servo Controller Boards & Chips

www.YostEngineering.com/ServoCenter

Yost Engineering, Inc.

1-888-395-9029

www.YostEngineering.com

What will your robot do?

It's up to you!