Interfacing the AT keyboard.

Why would you want to interface the Keyboard? The IBM keyboard can be a cheap alternative to a keyboard on a Microprocessor development system. Or maybe you want a remote terminal, just couple it with a LCD Module.

Maybe you have a RS-232 Barcode Scanner or other input devices, which you want to use with existing software which only allows you to key in numbers or letters. You could design yourself a little box to convert RS-232 into a Keyboard Transmission, making it transparent to the software.

An interfacing example is given showing the keyboard's protocols in action. This interfacing example uses a 68HC705J1A MCU to decode an IBM AT keyboard and output the ASCII equivalent of the key pressed at 9600 BPS.

Note that this page only deals with AT Keyboards. If you have any XT keyboards, you wish to interface, consider placing them in a museum. We will not deal with this type of keyboard in this document. XT Keyboards use a different protocol compared to the AT, thus code contained on this page will be incompatible.

PC Keyboard Theory

The IBM keyboard you most probably have sitting in front of you, sends scan codes to your computer. The scan codes tell your Keyboard Bios, what keys you have pressed or released. Take for example the 'A' Key. The 'A' key has a scan code of 1C (hex). When you press the 'A' key, your keyboard will send 1C down it's serial line. If you are still holding it down, for longer than it's typematic delay, another 1C will be sent. This keeps occurring until another key has been pressed, or if the 'A' key has been released.

However your keyboard will also send another code when the key has been released. Take the example of the 'A' key again, when released, the keyboard will send F0 (hex) to tell you that the key with the proceeding scan code has been released. It will then send 1C, so you know which key has been released.

Your keyboard only has one code for each key. It doesn't care it the shift key has been pressed. It will still send you the same code. It's up to your keyboard BIOS to determine this and take the appropriate action. Your keyboard doesn't even process the Num Lock, Caps Lock and Scroll Lock. When you press the Caps Lock for example, the keyboard

will send the scan code for the cap locks. It is then up to your keyboard BIOS to send a code to the keyboard to turn on the Caps lock LED.

Now there's 101 keys and 8 bits make 256 different combinations, thus you only need to send one byte per key, right?

Nop. Unfortunately a handful of the keys found on your keyboard are extended keys, and thus require two scan code. These keys are preceded by a E0 (hex). But it doesn't stop at two scan codes either. How about E1,14,77,E1,F0,14,F0,77! Now that can't be a valid scan code? Wrong again. It's happens to be sent when you press the Pause/break key. Don't ask me why they have to make it so long! Maybe they were having a bad day or something?

When an extended key has been released, it would be expect that F0 would be sent to tell you that a key has been released. Then you would expect E0, telling you it was an extended key followed by the scan code for the key pressed. However this is not the case. E0 is sent first, followed by F0, when an extended key has been released.

Keyboard Commands

Besides Scan codes, commands can also be sent to and from the keyboard. The following section details the function of these commands. By no means is this a complete list. These are only some of the more common commands.

Host Commands

These commands are sent by the Host to the Keyboard. The most common command would be the setting/resetting of the Status Indicators (i.e. the Num lock, Caps Lock & Scroll Lock LEDs). The more common and useful commands are shown below.

ED **Set Status LED's - This command can be used to turn on and off the Num Lock, Caps Lock & Scroll Lock LED's. After Sending ED, keyboard will reply with ACK (FA) and wait for another byte which determines their Status. Bit 0 controls the Scroll Lock, Bit 1 the Num Lock and Bit 2 the Caps lock. Bits 3 to 7 are ignored.**

EE Echo - Upon sending a Echo command to the Keyboard, the keyboard should reply with a Echo (EE)

F0 Set Scan Code Set. Upon Sending F0, keyboard will reply with ACK (FA) and wait for another byte, 01-03 which determines the Scan Code Used. Sending 00 as the second byte will return the Scan Code Set currently in Use

F3 Set Typematic Repeat Rate. Keyboard will Acknowledge command with FA and wait for second byte, which determines the Typematic Repeat Rate.

F4 Keyboard Enable - Clears the keyboards output buffer, enables Keyboard Scanning and returns an Acknowledgment.

F5 Keyboard Disable - Resets the keyboard, disables Keyboard Scanning and returns an Acknowledgment.

FE Resend - Upon receipt of the resend command the keyboard will re- transmit the last byte sent.

FF Reset - Resets the Keyboard.

Commands

Now if the Host Commands are send from the host to the keyboard, then the keyboard commands must be sent from the keyboard to host. If you think this way, you must be correct. Below details some of the commands which the keyboard can send.

FA Acknowledge

AA Power On Self Test Passed (BAT Completed)

EE See Echo Command (Host Commands)

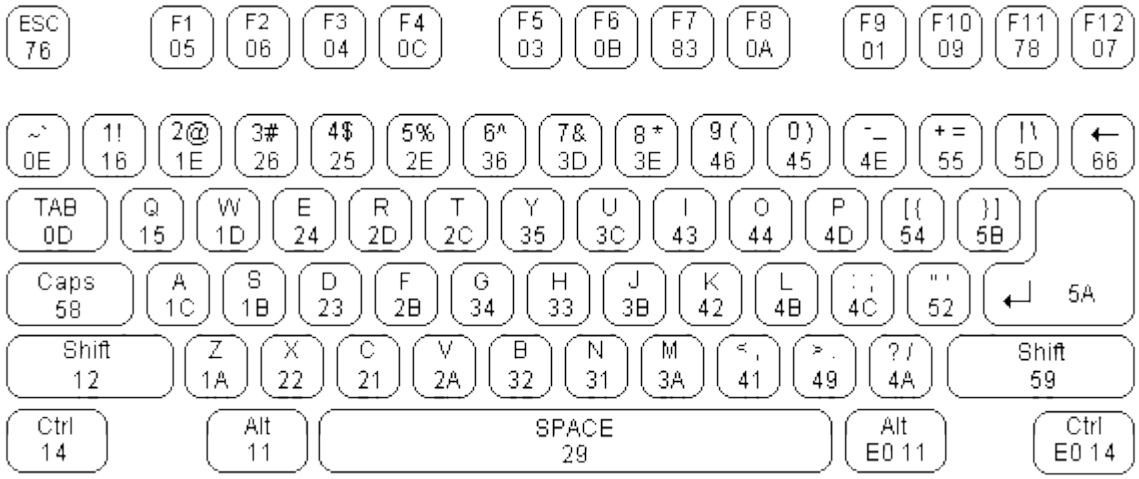
FE Resend - Upon receipt of the resend command the Host should re-transmit the last byte sent.

00 Error or Buffer Overflow

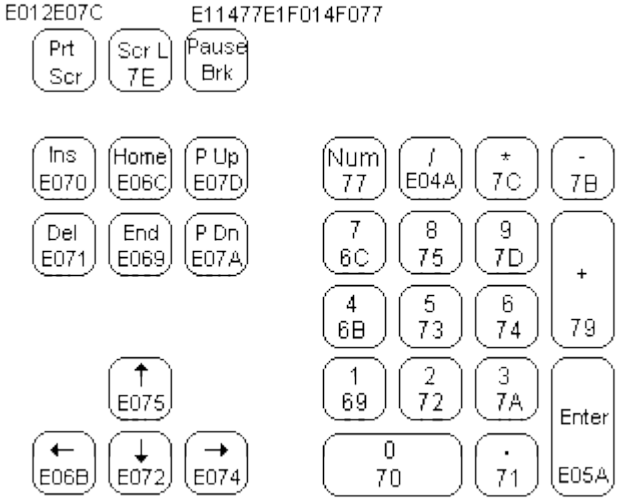
FF Error or Buffer Overflow

Scan Codes

The diagram below shows the Scan Code assigned to the individual keys. The Scan code is shown on the bottom of the key. E.g. The Scan Code for ESC is 76. All the scan codes are shown in Hex.

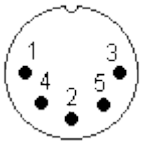


As you can see, the scan code assignments are quite random. In many cases the easiest way to convert the scan code to ASCII would be to use a look up table. Below is the scan codes for the extended keyboard & Numeric keypad.



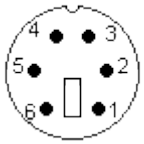
The Keyboard's Connector

The PC's AT Keyboard is connected to external equipment using four wires. These wires are shown below for the 5 Pin DIN Male Plug & PS/2 Plug.



5 Pin DIN

1. KBD Clock
2. KBD Data
3. N/C
4. GND
5. +5V (VCC)



PS/2

1. KBD Clock
2. GND
3. KBD Data
4. N/C
5. +5V (VCC)
6. N/C

A fifth wire can sometimes be found. This was once upon a time implemented as a Keyboard Reset, but today is left disconnected on AT Keyboards. Both the KBD Clock and KBD Data are Open Collector bi-directional I/O Lines. If desired, the Host can talk to the keyboard using these lines.

Note: Most keyboards are specified to drain a maximum 300mA. This will need to be considered when powering your devices

The Keyboard's Protocol

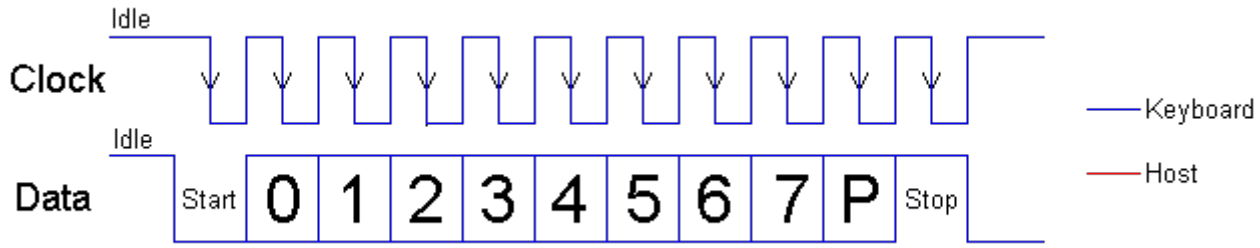
Keyboard to Host

As mentioned before, the PC's keyboard implements a bi-directional protocol. The keyboard can send data to the Host and the Host can send data to the Keyboard. The Host has the ultimate priority over direction. It can at anytime (although the not recommended) send a command to the keyboard.

The keyboard is free to send data to the host when both the KBD Data and KBD Clock lines are high (Idle). The KBD Clock line can be used as a Clear to Send line. If the host takes the KBD Clock line low, the keyboard will buffer any data until the KBD Clock is released, ie goes high. Should the Host take the KBD Data line low, then the keyboard will prepare to accept a command from the host.

The transmission of data in the forward direction, ie Keyboard to Host is done with a frame of 11 bits. The first bit is a Start Bit (Logic 0) followed

by 8 data bits (LSB First), one Parity Bit (Odd Parity) and a Stop Bit (Logic 1). Each bit should be read on the falling edge of the clock.

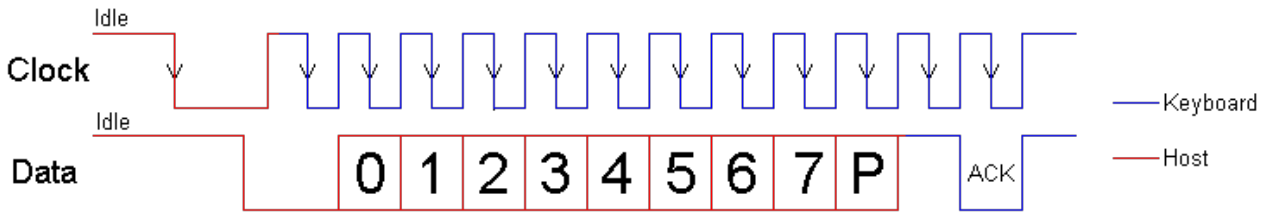


The above waveform represents a one byte transmission from the Keyboard. The keyboard may not generally change it's data line on the rising edge of the clock as shown in the diagram. The data line only has to be valid on the falling edge of the clock. The Keyboard will generate the clock. The frequency of the clock signal typically ranges from 20 to 30 Khz. The Least Significant Bit is always sent first.

Host to Keyboard

The Host to Keyboard Protocol is initiated by taking the KBD data line low. However to prevent the keyboard from sending data at the same time that you attempt to send the keyboard data, it is common to take the KBD Clock line low for more than 60us. This is more than one bit length. Then the KBD data line is taken low, while the KBD clock line is released.

The keyboard will start generating a clock signal on it's KBD clock line. This process can take up to 10mS. After the first falling edge has been detected, you can load the first data bit on the KBD Data line. This bit will be read into the keyboard on the next falling edge, after which you can place the next bit of data. This process is repeated for the 8 data bits. After the data bits come an Odd Parity Bit.



Once the Parity Bit has been sent and the KBD Data Line is in a idle (High) state for the next clock cycle, the keyboard will acknowledge the reception of the new data. The keyboard does this by taking the KBD Data line low for the next clock transition. If the KBD Data line is not idle after the 10th bit (Start, 8 Data bits + Parity), the keyboard will continue to send a KBD Clock signal until the KBD Data line becomes idle.

Interfacing Example - Keyboard to ASCII Decoder

Normally in this series of web pages, we connect something to the PC, to demonstrate the protocols at work. However this poses a problem with the keyboard. What could be possibly want to send to the computer via the keyboard interface?

Straight away any devious minds would be going, why not a little box, which generates passwords!. It could keep sending characters to the computer until it finds the right sequence. Well I'm not going to encourage what could possibly be illegal practices.

In fact a reasonably useful example will be given using a 68HC705J1A single chip microcontroller. We will get it to read the data from the keyboard, convert the scan codes into ASCII and send it out in RS-232 format at 9600 BPS. However we won't stop here, you will want to see the bi-directional use of the KBD Clock & Data lines, thus we will use the keyboards status LEDS, Num Lock, Caps Lock and Scroll Lock.

This can be used for quite a wide range of things. Teamed up with a reasonably sized 4 line x 40 character LCD panel, you could have yourself a little portable terminal. Or you could use it with a microcontroller development system. The 68HC705J1A in a One Time Programmable (OTP) is only a fraction of the cost of a 74C922 keyboard decoder chip, which only decodes a 4 x 4 matrix keypad to binary.

The keyboard doesn't need to be expensive either. Most people have many old keyboards floating around the place. If it's an AT Keyboard, then use it (XT keyboards will not work with this program.) If we ever see the introduction of USB keyboards, then there could be many redundant AT keyboards just waiting for you to hook them up.

Features

Before we start with the technical aspects of the project, the salesman in me wants to tell you about the features packed into the 998 bytes of code.

•Use of the keyboard's bi-directional protocol allowing the status of the Num Lock, Caps Lock and Scroll Lock to be displayed on the Keyboards LEDs.

•External Reset Line activated by ALT-CTRL-DEL. If you are using it with a Microcontroler development system, you can reset the MCU with the keyboard. *I've always wanted to be able to use the three fingered solute on the HC11!*

•Scroll Lock and Num Lock toggles two Parallel Port Pins on the HC705. This can be used to turn things on or off, Select Memory Pages, Operating Systems etc

•"ALTDEC" or what I call the Direct Decimal Enter Routine. Just like using a PC, when you enter a decimal number when holding down one of the ALT keys the number is sent as binary to the target system. E.g. If you press and hold down ALT, then type in 255 and release ALT, the value FF (Hex) will be sent to the system. *Note. Unlike the PC, you can use both the numeric keypad or the numbers along the top of the keyboard.*

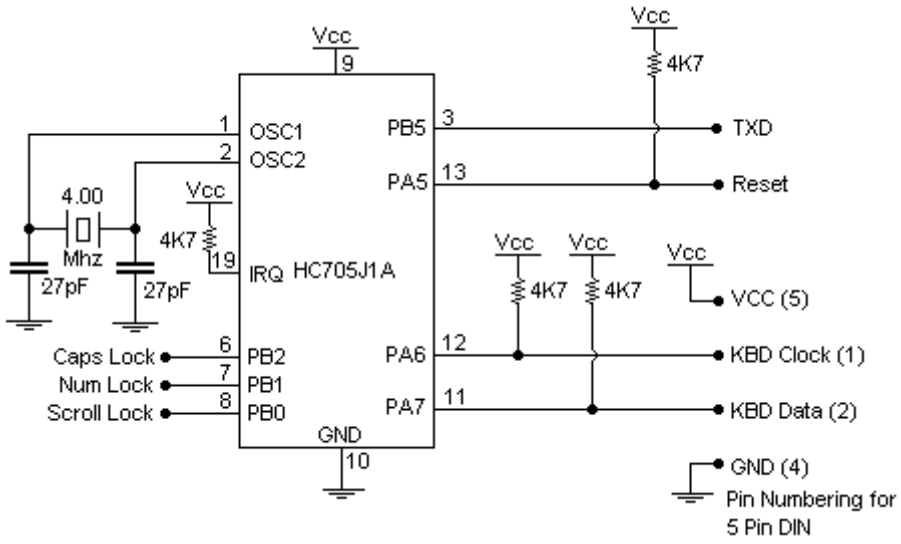
•"CTRLHEX" or you guessed it, Direct Hexadecimal Enter Routine. This function is not found with the PC. If you hold CTRL down, you can enter a Hexadecimal number. Just the thing for Development Systems or even debugging RS-232 Comms?

•Output is in ASCII using a RS-232 format at 9600 BPS. If using it with a development System, you can tap it in after the RS-232

Line Transceivers to save you a few dollars on RS-232 Level Converters.

Schematic & Hardware

The schematic below, shows the general connections of the keyboard to the HC705.



The TXD pin, while it transmits in RS-232 format, is not at RS-232 Voltage Levels. If you want to connect it to RS-232 Devices then you will need to attach a RS-232 Level Converter of some kind. If you are using it with a development system, you can bypass both RS-232 Level Converters and connect it directly to the RXD pin of the MCU. However the keyboard can't be a direct replacement for a terminal on a development system, unless you want to type in your code each time! You may want to place a jumper or switch inline to switch between your RS-232 Port and the Keyboard.

The Keyboard requires open collector/open drain outputs. This is achieved by using the Data Direction Register (DDR). A zero is written to the port which is internally latched. The DDR is then used to toggle the line from logic 0 to high impedance. If the port pin is an output, a logic zero will be present on the pin, if the port is set to be an input, the port will be high impedance which is pulled high by the external resistors.

The circuit is designed to run on a 4Mhz crystal (2Mhz Bus Speed). The timing for the RS-232 transmission is based on the bus speed, thus this

crystal has to be 4 Mhz. If you are lucky enough to have a 4 Mhz E Clock on your development system you can use it.

The power supply can also create a slight problem. A standard keyboard can drain about 300mA max, thus it would be recommended to use it's own regulator rather than taking a supply from elsewhere. Decoupling capacitors are not shown on the general schematic but are implied for reliable operation. Consult your MC68HC705J1A Technical Data Manual for more Information.

Reading Bytes from the Keyboard.

Now it is time to look at the code. I cannot include a description of all the code in this article. The list file is just on 19 pages. Most of it (hopefully) is easy to follow. (Just like other good code, count the number of spelling errors while you are at it!)

Remember the KBD Clock line? If you take it low, the keyboard will buffer any keys pressed. The Keyboard will only attempt to send when both the Data and Clock lines are idle (high). As it can take considerable time to decode the keys pressed, we must stop the keyboard from sending data. If not, some of the data may be lost or corrupted.

```
Receive  ldx     #08                ;Number of Bits
         clr     PAR                ;Clear Parity Register
         bclr    clk,DDRA           ;Clear to Send

         brset   clk,PORTA,*        ;wait on idle Clock
         brset   data,PORTA,Receive ;False Start Bit, Restart
```

The program, will keep the KBD Clock line low, unless it is ready to accept data. We will use a loop to retrieve the data bits from the keyboard, thus we will load index register X with the number of bits be want to receive. PAR will be used to verify the parity bit at the end of the transmission. We must clear this first.

We can then place the KBD Clock line in the idle state so that the keyboard will start transmitting data if a key has been pressed. The program then loops while the clock line is Idle. If the KBD clock goes low, the loop is broken and the KBD Data pin is read. This should be the

start bit which should be low. If not we branch to the start of the receive routine and try again.

```
Recdata  ror     byte
         jsr     highlow          ;Wait for high to low Transition
         brset   data,PORTA,Recset
         bclr    7,byte
         jmp     Recnext
Recset   bset    7,byte
         inc     PAR
Recnext  decx
         bne     Recdata          ;Loop until 8 bits been received
```

Once the start bit has been detected, the 8 data bits must follow. The data is only valid on the falling edge of the clock. The subroutine highlow shown below will wait for the falling edge of the clock.

```
highlow brclr   clk,PORTA,*      ;Loop until Clk High
        brset   clk,PORTA,*      ;Loop until Clk Low
        rts
```

After the falling edge we can read the level of the KBD Data line. If it is high we can set the MSbit of the byte or if it is clear, we can clear it. You will notice if the bit is set, we also increment PAR. This keeps track of the number of 1's in the byte and thus can be used to verify the Parity Bit. Index register X is decremented as we have read a bit. It then repeats the above process, until the entire 8 bits have been read.

```
        lda         PORTA        ; MSb is Parity.
        rola                     ; Shift MSbit to LSbit.
        rola                     ; thru carry
        eor         PAR
        and         #$01
        beq         r_error
```

After the 8 data bits, comes the dreaded parity bit. We could ignore it if we wanted to, but we may as well do something about it. We have been keeping a tally of the number of 1's in PAR. The keyboard uses odd parity, thus the parity bit should be the complement of the LSbit in memory location, PAR. By exclusive OR-ing PAR with the Parity Bit, we get a 1 if both the bits are different. I.e a '1' if the parity bit checks out.

As we are only interested in the LSbit we can quite happy XOR the accumulator with PAR. Then we single out the LSb using the AND

function. If the resultant is zero, then a parity error has occurred and the program branches to r_error.

```
            jsr     highlow
            brclr   data,PORTA,r_error      ;Stop Bit Detection

            bset    clk,DDRA                ;Prevent Keyboard from sending
data
                                            ;(Clear to Send)
            rts
```

After the Parity Bits comes the Stop Bit. Once again we can ignore it if we desire. However we have chosen to branch to an error routine if this occurs. The stop bit should be set, thus an error occurs when it is clear.

```
  r_error lda     #$FE                ;Resend
            sta     byte
            jsr     Transmit
            jmp     Receive             ;Try again
```

What you do as error handling is up to you. In most cases it will never be executed. In fact I don't yet know if the above error handling routine works. I need to program another HC705 to send a false parity bit. I've tried it out in close proximity to the Washing Machine, but I really need a controlled source!

When an error occurs in the Parity or Stop Bit we should assume that the rest of the byte could have errors as well. We could ignore the error and process the received byte, but it could have unexpected results. Instead the keyboard has a resend command. If we issue a resend (FE) to the keyboard, the keyboard should send the byte back again. This is what occurs here.

You may notice that we branch to the error routine which transmits a resend command straight away, without waiting for the corrupt transmission to finish. This is not a problem, as the keyboard considers any transmission to be successful, if the 10th bit is sent, i.e. the parity bit. If we interrupt the transmission before the parity bit is sent, the keyboard will place the current byte in it's buffer for later transmission.

Reading a byte doesn't really require bi-directional data and clock lines. If you can process the byte fast enough then no handshaking (RTS) is required. This means you no longer need to fiddle with the Data Direction Register. I have successfully done this with the HC705,

outputting only scan codes on a parallel bus. But as you can imagine, you must be quick in order to catch the next transmission.

Writing Bytes to the Keyboard.

The following routine given here is a generic one which can be used for your own purposes. During normal execution of this program the KBD clock line should be low, to prevent data being sent when the MCU isn't ready for it. However in this example, we take low the KBD clock line and wait for the 64uS which is pointless as the line is already low and has been like this for quite some time, since the end of the last transmission or reception.

```
transmit ldx     #$08                ;8 Data Bits
         bset    clk,DDRA                ;Set Clock Low
         lda     #$13                ;Delay 64uS
         jsr     delay
         clra                    ;Clear Parity Register
         bset    data,DDRA               ;Set Data Low
         bclr    clk,DDRA            ;Release Clock Line
         jsr     highlow
```

The program then initiates the Host to Keyboard transmission by taking the KBD data line low and releasing the KBD clock line. We must then wait for a high to low transition on the KBD clock, before we load the first bit on the KBD data line.

```
loop     ror     byte
         bcs     mark
space    bset    data,DDRA               ; Clear Bit
         jmp     next
mark     bclr    data,DDRA               ; Clear Bit
         inca                    ; Parity Calculation
next     jsr     highlow             ; Wait for high to low transition
         decx
         bne     loop
```

The loading of the individual bits on the KBD data line is done in very similar fashion to the read cycle. The X register is used to keep track of the number of bits sent. Also simular to the read cycle, we increment the accumulator so we can calculate the parity bit later on.

```
            and     #$01
            bne     clr_par
set_par     bclr    data,DDRA
            jmp     tr_ackn
clr_par     bset    data,DDRA
tr_ackn     jsr     highlow
```

After the data bits have been sent, it is now time to send the parity bit. Unlike the read cycle, we can't ignore the parity bit. If we do the keyboard will issue a resend (FE) command if the parity bit is incorrect, a 50% probability!

```
            bclr    data,DDRA           ;Release Data Line
            jsr     highlow
            brset   data,PORTA,error    ;Check for Ack
            brclr   clk,PORTA,*         ;Wait for idle line

            bset    clk,DDRA            ;Prevent Keyboard from sending
data
                                        ;(Clear to Send)
            rts
```

Once the Parity bit has been set and the falling edge of the KBD clock detected, we must release the KBD data line, and wait for another falling edge of the KBD clock to see if the Keyboard has acknowledged the byte. The keyboard does this by pulling the KBD data line low. If it is not low, then the program branches to an error handler. If all has been successful, the MCU pulls down the KBD clock, to prevent it from transmitting.

```
error   lda     #$FF    ;Reset
            sta     byte
            jsr     transmit
            rts
```

We have taken a harsher approach to handing any transmit errors. Ideally we should wait for the keyboard to send a resend command and then retransmit the byte. However what we have done is to issue a reset to the keyboard. So far I've never had an error, however if this starts to become a problem, then a better error handler could be written.