AVR313: Interfacing the PC AT Keyboard

8-bit AVR® Microcontroller

Application Note

Features

- Interfacing Standard PC AT Keyboards
- Requires Only Two I/O Pins. One of them must be an External Interrupt Pin
- No Extra Hardware Required
- Complete Example in C, Implementing a Keyboard to Serial Converter

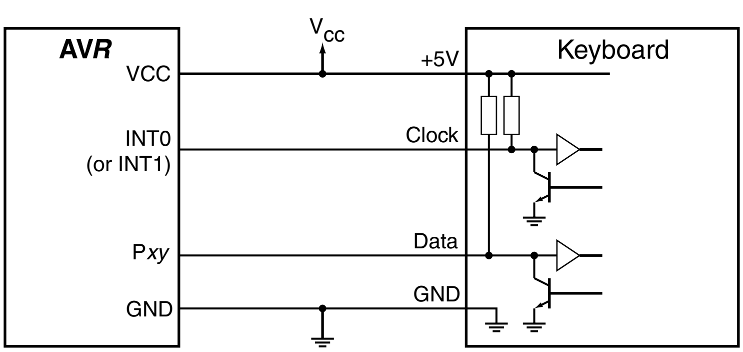
Introduction

Most microcontrollers requires some kind of a human interface. This application note describes one way of doing this using a standard PC AT keyboard.

The Physical Interface

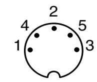
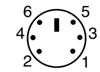
The physical interface between the keyboard and the host is shown in Figure 1. Two signal lines are used, clock and data. The signal lines are open connector, with pullup resistors located in the keyboard. This allows either the keyboard or the host system to force a line to low level. Two connector types are available, the 5-pin DIN connector of “5D” type, and the smaller six-pin mini-DIN. The pin assignments are shown in Table 1.

Figure 1. The Interface.



Rev. 1235B–AVR–05/02

Table 1. AT Keyboard Connector Pin Assignments

AT Computer		
Signals	**DIN41524, Female at Computer, 5-pin DIN 180^{o}**	**6-pin Mini DIN PS2 Style Female at Computer**
Clock	1	5
Data	2	1
nc	3	2,6
GND	4	3
+5V	5	4
Shield	Shell	Shell

Timing

The timing for the data transferred from the keyboard to the host is shown in Figure 2. The protocol is: one start bit (always 0), eight data bits, one odd parity bit and one stop bit (always 1). The data is valid during the low period of the clock pulse. The keyboard is generating the clock signal, and the clock pulses are typically 30-50 µs low and 30-50 µs high.

The host system can send commands to the keyboard by forcing the clock line low. It then pulls the data line low (the start bit). Now, the clock line must be released. The keyboard will count 10 clock pulses. The data line must be set up to the right level by the host before the trailing edge of the clock pulse. After the tenth bit, the keyboard checks for a high level on the data line (the stop bit), and if it is high, it forces it low. This tells the host that the data is received by the keyboard. The software in this design note will not send any commands to the keyboard.

Scan Codes

The AT keyboard has a scan code associated with each key. When a key is pressed, this code is transmitted. If a key is held down for a while, it starts repeating. The repeat rate is typically 10 per second. When a key is released, a "break" code ($F0) is transmitted followed by the key scan code. For most of the keys, the scan code is one byte. Some keys like the *Home*, *Insert* and *Delete* keys have an extended scan code, from two to five bytes. The first byte is always $E0. This is also true for the "break" sequence, e.g., E0 F0 xx…

AT keyboards are capable of handling three sets of scan codes, where set two is default. This example will only use set two.

The Software

The code supplied with this application note is a simple keyboard to RS-232 interface. The scan codes received from the keyboard are translated into appropriate ASCII characters and transmitted by the UART. The source code is written in C, and is easily modified and adaptable to all AVR microconrollers with SRAM.

Note: The linkerfile (AVR313.xcl) included in the software archive has to be included instead of the standard linkerfile. This is done from the include menu under XLINK – Options. The linker file applies to AT90S8515 only.

The Algorithm

Keyboard reception is handled by the interrupt function **INT0_interrupt**. The reception will operate independent of the rest of the program.

The algorithm is quite simple: Store the value of the data line at the leading edge of the clock pulse. This is easily handled if the clock line is connected to the INT0 or INT1 pin. The interrupt function will be executed at every edge of the clock cycle, and data will be stored at the falling edge. After all bits are received, the data can be decoded. This is done by calling the **decode** function. For character keys, this function will store an ASCII character in a buffer. It will take into account if the shift key is held down when a key is pressed. Other keys like function keys, navigation keys (arrow keys, page up/down keys etc.) and modifier keys like Ctrl and Alt are ignored.

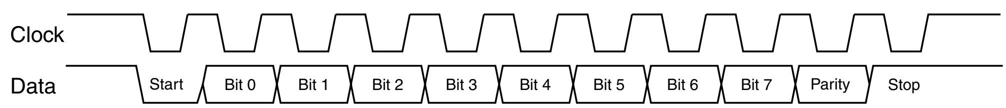
The mapping from scan codes to ascii characters are handled with table look-ups, one table for shifted characters and one for un-shifted.

Modifications and Improvements

If the host falls out of sync with the keyboard, all subsequent data received will be wrong. One way to solve this is to use a time out. If 11 bits are not received within 1.5 ms, some error have occurred. The bit counter should be reset and the faulty data discarded.

If keyboard parameters like typematic rate and delay are to be set, data must be sent to the keyboard. This can be done as described earlier. For the commands, see the keyboard manufacturer's specifications.

Figure 2. Timing for Keyboard to Host Transfer



Main.c

```
#include <pgmspace.h>
#include <stdio.h>
#include <stdlib.h>
#include "io8515.h"

#include "serial.h"
#include "gpr.h"
#include "kb.h"

void main(void)
{
  unsigned char key;

  init_uart();                    // Initializes the UART transmit buffer
  init_kb();                      // Initialize keyboard reception

  while(1)
  {
    key=getchar();
    putchar(key);
    delay(100);
  }
}
```

Low_level_init.c

```
#include <ina90.h>
#include <io8515.h>

int __low_level_init(void)
{
    UBRR = 12;          // 19200bps @ 4 MHz
    UCR  = 0x08;        // TX enable
    GIMSK= 0x40;        // Enable INT0 interrupt

    _SEI();
    return 1;
}
```

Serial.c

```
#include <stdio.h>
#include <pgmspace.h>
#include <io8515.h>                 /* SFR declarations */
#include "serial.h"

#define ESC 0x1b
#define BUFF_SIZE 64

flash char CLR[] = {ESC, '[','H', ESC, '[', '2', 'J',0};

unsigned char UART_buffer[BUFF_SIZE];
unsigned char *inptr, *outptr;
unsigned char buff_cnt;

void init_uart(void)
{
    inptr =  UART_buffer;
    outptr = UART_buffer;
    buff_cnt = 0;
}

void clr(void)
{
    puts_P(CLR);                              // Send a 'clear screen' to a
VT100 terminal
}

int putchar(int c)
{
    if (buff_cnt<BUFF_SIZE)
    {
        *inptr = c;                           // Put character into buffer
        inptr++;                              // Increment pointer

        buff_cnt++;

        if (inptr >= UART_buffer + BUFF_SIZE)   // Pointer wrapping
```

```
            inptr = UART_buffer;

        UCR = 0x28;                                 // Enable UART Data register
                                                    // empty interrupt

        return 1;
    } else {
        return 0;                                   // Buffer is full
    }

}

   // Interrupt driven transmitter

interrupt [UART_UDRE_vect] void UART_UDRE_interrupt(void)
{
    UDR = *outptr;                                  // Send next byte
    outptr++;                                       // Increment pointer

    if (outptr >= UART_buffer + BUFF_SIZE)          // Pointer wrapping
        outptr = UART_buffer;

    if(--buff_cnt == 0)                             // If buffer is empty:
        UCR = UCR && (1<<UDRIE);                    //    disabled interrupt
}
```

Kb.c

```
#include <pgmspace.h>
#include "kb.h"
#include "serial.h"
#include "gpr.h"

#include "scancodes.h"

#define BUFF_SIZE 64

unsigned char edge, bitcount;// 0 = neg.  1 = pos.

unsigned char kb_buffer[BUFF_SIZE];
unsigned char *inpt, *outpt;
unsigned char buffcnt;


void init_kb(void)
{
    inpt =  kb_buffer;// Initialize buffer
    outpt = kb_buffer;
    buffcnt = 0;

  MCUCR = 2;  // INT0 interrupt on falling edge
```

```
  edge = 0;  // 0 = falling edge  1 = rising edge
  bitcount = 11;
}

interrupt [INT0_vect] void INT0_interrupt(void)
{
  static unsigned char data;// Holds the received scan code

  if (!edge) // Routine entered at falling edge
  {
    if(bitcount < 11 && bitcount > 2)// Bit 3 to 10 is data. Parity bit,
    {           // start and stop bits are ignored.
      data = (data >> 1);
      if(PIND & 8)
        data = data | 0x80;// Store a '1'
    }

    MCUCR = 3;// Set interrupt on rising edge
    edge = 1;

  } else {   // Routine entered at rising edge

    MCUCR = 2;// Set interrupt on falling edge
    edge = 0;

    if(--bitcount == 0)// All bits received
    {
      decode(data);
      bitcount = 11;
    }
  }
}


void decode(unsigned char sc)
{
  static unsigned char is_up=0, shift = 0, mode = 0;
  unsigned char i;
  if (!is_up)// Last data received was the up-key identifier
  {
    switch (sc)
    {
      case 0xF0 :// The up-key identifier
      is_up = 1;
      break;

      case 0x12 :// Left SHIFT
      shift = 1;
      break;
```

```
    case 0x59 :// Right SHIFT
    shift = 1;
    break;

    case 0x05 :// F1
    if(mode == 0)
      mode = 1;// Enter scan code mode
    if(mode == 2)
      mode = 3;// Leave scan code mode
    break;

    default:
    if(mode == 0 || mode == 3)// If ASCII mode
    {
      if(!shift)// If shift not pressed,
      {     // do a table look-up
        for(i = 0; unshifted[i][0]!=sc && unshifted[i][0]; i++);
        if (unshifted[i][0] == sc) {
          put_kbbuff(unshifted[i][1]);
        }
      } else {// If shift pressed
        for(i = 0; shifted[i][0]!=sc && shifted[i][0]; i++);
        if (shifted[i][0] == sc) {
          put_kbbuff(shifted[i][1]);
        }
      }
    } else{ // Scan code mode
      print_hexbyte(sc);// Print scan code
      put_kbbuff(' ');
      put_kbbuff(' ');
    }
    break;
  }
} else {
  is_up = 0;// Two 0xF0 in a row not allowed
  switch (sc)
  {
    case 0x12 :// Left SHIFT
    shift = 0;
    break;

    case 0x59 :// Right SHIFT
    shift = 0;
    break;

    case 0x05 :// F1
    if(mode == 1)
      mode = 2;
    if(mode == 3)
      mode = 0;
```

```
      break;
      case 0x06 :// F2
      clr();
      break;
    }
  }
}

void put_kbbuff(unsigned char c)
{
  if (buffcnt<BUFF_SIZE)// If buffer not full
  {
    *inpt = c;// Put character into buffer
    inpt++;    // Increment pointer

    buffcnt++;

    if (inpt >= kb_buffer + BUFF_SIZE)// Pointer wrapping
      inpt = kb_buffer;
    }
}

int getchar(void)
{
  int byte;
  while(buffcnt == 0);// Wait for data

  byte = *outpt;// Get byte
  outpt++;    // Increment pointer

  if (outpt >= kb_buffer + BUFF_SIZE)// Pointer wrapping
    outpt = kb_buffer;

  buffcnt--; // Decrement buffer count

  return byte;
}
```

Gpr.c

```
#include "gpr.h"

void print_hexbyte(unsigned char i)
{
    unsigned char h, l;

    h = i & 0xF0;               // High nibble
    h = h>>4;
    h = h + '0';

    if (h > '9')
        h = h + 7;

    l = (i & 0x0F)+'0';         // Low nibble
    if (l > '9')
        l = l + 7;


    putchar(h);
    putchar(l);
}


void delay(char d)
{
    char i,j,k;
    for(i=0; i<d; i++)
        for(j=0; j<40; j++)
            for(k=0; k<176; k++);
}
```

Pindefs.h

```
//*********************
// Pin definition file
//*********************


// Keyboard konnections
#define PIN_KB  PIND
#define PORT_KB PORTD
#define CLOCK   2
#define DATAPIN 3
```

Scancodes.h

```
// Unshifted characters
flash unsigned char unshifted[][2] = {
0x0d,9,
0x0e,'|',
0x15,'q',
0x16,'1',
0x1a,'z',
0x1b,'s',
0x1c,'a',
0x1d,'w',
0x1e,'2',
0x21,'c',
0x22,'x',
0x23,'d',
0x24,'e',
0x25,'4',
0x26,'3',
0x29,' ',
0x2a,'v',
0x2b,'f',
0x2c,'t',
0x2d,'r',
0x2e,'5',
0x31,'n',
0x32,'b',
0x33,'h',
0x34,'g',
0x35,'y',
0x36,'6',
0x39,',',
0x3a,'m',
0x3b,'j',
0x3c,'u',
0x3d,'7',
0x3e,'8',
0x41,',',
0x42,'k',
0x43,'i',
0x44,'o',
0x45,'0',
0x46,'9',
0x49,'.',
0x4a,'-',
0x4b,'l',
0x4c,'ø',
0x4d,'p',
0x4e,'+',
0x52,'æ',
0x54,'å',
0x55,'\\',
```

```
0x5a,13,
0x5b,'¨',
0x5d,'\'',
0x61,'<',
0x66,8,
0x69,'1',
0x6b,'4',
0x6c,'7',
0x70,'0',
0x71,',',
0x72,'2',
0x73,'5',
0x74,'6',
0x75,'8',
0x79,'+',
0x7a,'3',
0x7b,'-',
0x7c,'*',
0x7d,'9',
0,0
};
// Shifted characters
flash unsigned char shifted[][2] = {
0x0d,9,
0x0e,'§',
0x15,'Q',
0x16,'!',
0x1a,'Z',
0x1b,'S',
0x1c,'A',
0x1d,'W',
0x1e,'"',
0x21,'C',
0x22,'X',
0x23,'D',
0x24,'E',
0x25,'¤',
0x26,'#',
0x29,' ',
0x2a,'V',
0x2b,'F',
0x2c,'T',
0x2d,'R',
0x2e,'%',
0x31,'N',
0x32,'B',
0x33,'H',
0x34,'G',
0x35,'Y',
0x36,'&',
```

```
0x39,’L’,
0x3a,’M’,
0x3b,’J’,
0x3c,’U’,
0x3d,’/’,
0x3e,’(’,
0x41,’;’,
0x42,’K’,
0x43,’I’,
0x44,’O’,
0x45,’=’,
0x46,’)’,
0x49,’:’,
0x4a,’_’,
0x4b,’L’,
0x4c,'Ø',
0x4d,'P',
0x4e,'?',
0x52,'Æ',
0x54,'Å',
0x55,'`',
0x5a,13,
0x5b,'^',
0x5d,'*',
0x61,'>',
0x66,8,
0x69,'1',
0x6b,'4',
0x6c,'7',
0x70,'0',
0x71,',',
0x72,'2',
0x73,'5',
0x74,'6',
0x75,'8',
0x79,'+',
0x7a,'3',
0x7b,'-',
0x7c,'*',
0x7d,'9',
0,0
};
```

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

1235B–AVR–05/02 0M