Locus Engineering Inc. AN102

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

Abstract

The PS/2 keyboard is an ideal low cost data entry device for embedded microcontroller applications. Used with the Locus Engineering E1115B PS/2 Keyboard to ASCII Converter which produces single bytes on the make of every keystroke, this application note explains how to parse this simpler output into commands and data. This note assumes single byte commands with single or multi-byte data. 8051 architecture assembly code will be used for the example although the techniques are adaptable to any microcontroller. This note also assumes basic programming skills such as interrupt handling and display routines.

The E1115B PS/2 Keyboard to ASCII Converter is an easy to use device for converting the multi-byte keyboard output into more manageable single bytes. The converter offloads the process intensive monitoring and decoding of complex scancodes from the host microcontroller, thereby saving on resources and program memory. The module produces both a serial data output which can be converted to RS-232 levels, and a ~100KHz clocked data output. The serial data rate can be either 115.2 or 57.6Kbaud. The serial output requires a single serial port pin while the clocked data output requires three pins.

In this example of parsing commands and data from a PS/2 keyboard, let the function keys F1 to F8 represent the commands, and the numbers 0-9 and letters a-f and A-F represent the data for hexadecimal numbers. Assume the serial port on the host microcontroller will be used to receive the byte from the E1115B converter. If no serial port is available on the host microcontroller, three port pins on the host microcontroller can be used to receive the 100KHz clocked serial data. The clocked data port interface is explained in the E1115B datasheet at www.CHiPdesign.ca.

The parsing procedure is to first receive a byte from the E1115B converter from the host microcontroller's serial port and store the received byte in a register. The keystroke is then validated and this is implemented using a 256 byte look-up table. This table provides one output value for every one of the 256 possible input values. All unused keys return a null value such as zero or 0x00 in hexadecimal while the “n” used keys such as for data and commands are made to return a value from 1 to “n”. The values from 1 to “n” and zero are chosen because they are used as input values to a second look-up table, so they need to be consecutive and start from zero. Thus there will be 256-n null output values of 0x00 for the unused keys and “n” output values between 0x01 and 0xn for the used keys representing the command routines and the data routines.

The second table provides “n+1” addresses which point to the actual command and data routines as well as the null routine.

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

In detail, the parsing procedure is shown in Table 1.

The E1115B output codes (all values are in hexadecimal) for the example keys are as follows:

Table 1. E1115B Converter Outputs vs. Used PS/2 Keyboard Inputs

PS/2 Key	Output		PS/2 Key	Output		PS/2 Key	Output
F1	0xF1		0	0x30		8	0x38
F2	0xF2		1	0x31		9	0x39
F3	0xF3		2	0x32		a, A	0x61, 0x41
F4	0xF4		3	0x33		b, B	0x62, 0x42
F5	0xF5		4	0x34		c, C	0x63, 0x43
F6	0xF6		5	0x35		d, D	0x64, 0x44
F7	0xF7		6	0x36		e, E	0x65, 0x45
F8	0xF8		7	0x37		f, F	0x66, 0x46

Note that both lower case “a” to “f” and upper case “A” to “F” letter keys are used as well as the function keys F1 to F8. All other keys are unused.

The microcontroller's serial port generates an interrupt when a byte is received from the E1115B converter. The interrupt service routine then clears the serial interrupt flag, reads the serial port receive register, and stores the byte at a register in the microcontroller's RAM. In this example, the register can be called RXREG.

Once a byte has been saved, it needs to be validated to separate the unused keys from the used keys. Unused keys should not generate any response other than return to wait for the next keystroke. Used keys should generate a response as defined by the user. Keys are validated by a look-up table which returns zero or null values for unused keys and non-zero codes for used keys. A second look-up table is used to generate the key routine addresses from the first table's results. The two table method is simpler to understand and generally takes less program memory space than a single 512 byte table.

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

The first look-up table will convert all the valid keys into an output from 1 to "n", and all the unused keys into an output of "0". Thus the function keys F1 to F8 are converted to 0x01 to 0x08; the numbers 0 to 9 are converted to 0x09 to 0x12, and the letters "a" to "f" and "A" to "F" are converted to 0x13 to 0x18. Thus there are 30 valid keys producing 24 valid codes from 0x01 to 0x18. All other keys will return the null value 0x00. The numbers 0 and 1 to "n" are selected so that the second look-up table is sequential and as compact as possible. The first look-up table values are shown in Table 2.

Table 2. First Look-Up Table Output vs. E1115B Converter Output as Input

Output								Input
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	;0x00-0x07
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	;0x08-0x0F
								...
0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	;0x30-0x37
0x11	0x12	0x00	0x00	0x00	0x00	0x00	0x00	;0x38-0x3F
								...
0x00	0x13	0x14	0x15	0x16	0x17	0x18	0x00	;0x40-0x47
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	;0x48-0x4F
								...
0x00	0x13	0x14	0x15	0x16	0x17	0x18	0x00	;0x60-0x67
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	;0x68-0x6F
								...
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	;0xF0-0xF7
0x08	0x00	0x00	0x00	0x00	0x00	0x00	0x00	;0xF8-0xFF

Each data key has its own routine. As an example, PS/2 keyboard letter key "D" returns a code of 0x44 from the E1115B converter. Applying this as an input to the table returns an output value of 0x16 as does the lower case "d" whose E1115B code is 0x64. Command key F8 returns a code of 0xF8 which returns a table value of 0x08. Note that most of the null output rows are not included in the table for brevity. The table returns 24 valid outputs and the remaining 232 unused inputs return the null output value 0x00.

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

The following '8051 code uses the value of RXREG and the base address LUT1 to return a byte into register LUT1OUT:

```
MOV DPTR, #LUT1   ;load base address to data pointer
MOV A, RXREG      ;load offset to accumulator
MOVC @A, A+DPTR   ;read look-up table value at address (base + offset)
MOV LUT1OUT, A    ;store look-up table 1 output
```

Once the first look-up table has been read, the result is used as an input to the second look-up table which returns the address of one of the data , command, or null routines. As these are addresses within the program memory, they are 16 bits or two bytes. Since there are two bytes for each input value, the table size is 2(n+1) bytes where "n" is the number of command and data routines and the null routine being the "1". To point to a routine, the input value is doubled and then added to the second look-up table's start address. The first value returned is the most significant byte, and the second value returned is the least significant byte of the address for the data or command routine. These two bytes are loaded into the Program Counter and the program jumps to the routine corresponding to the keystroke.

The second look-up table values are defined as shown in Table 3.

Table 3. 2nd Look-Up Table Routine Address Output vs. Validated Key Input.

Output								Input
KNL	KC1	KC2	KC3	KC4	KC5	KC6	KC7	0x00-0x07
KC8	KD0	KD1	KD2	KD3	KD4	KD5	KD6	0x08-0x0F
KD7	KD8	KD9	KDA	KDB	KDC	KDD	KDE	0x10-0x17
KDF	KNL	KNL	KNL	KNL	KNL	KNL	KNL	0x18-0x1F

Input values include 0x00 for the null keys, and 1 to "n" for the valid keys. Note the last row is padded with null outputs. This table does not use numerical values; instead it uses the names of the routines servicing the keys; the names are also 16 bit addresses represented as two bytes. The value "KNL" represents the location of the null service routine, wherever it may be in the program code. Likewise, the value "KC7" represents the location of the service routine for the command key F7. The value "KDC" represents the location of the service routine for the data keys "c" or "C".

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

In this example, the keystroke routines either do nothing, store data, or perform commands. Unused keys invoke the null routine which returns to check for the next keyboard entry. The data routines might convert the data byte to hexadecimal form, and then shift the nibble into a selected buffer where it will be available for the next command. Note that in this example the data is stored as two nibbles per byte. This is a more useful method of storing data than one nibble per byte because arithmetic operations can be more easily performed. The command routines typically "do" something usually defined by the application; this could be adding two numbers, navigating to a new screen, selecting a value, or entering a value. In the example to follow, an addition of two numbers using Reverse Polish Notation or RPN is demonstrated. In RPN, the first value is typed in followed by the Enter key, and then the second value is typed in followed by the operator; in this case the Add key. RPN is easy to implement compared to algebraic entry and does away with the requirement for parentheses.

The buffer used to store data is a set of consecutive registers in the microcontroller's RAM. The data buffer size is defined by the largest number of digits needed for an input. If a 16 bit address was needed to be entered, four nibbles or two bytes would be needed whereas a 64 bit number would need eight bytes.

More than one buffer may be used so that operations on several numbers can be executed. For example, one command may add two numbers so two data buffers would be needed for the input values and possibly a third for the output value. Initially the first data buffer would be selected and the first data would be stored in it. A command key defined as "Enter" would then select the second buffer for the second data to be stored in it. A second command key defined as "Add" would then add the data in the two buffers, store the result in a third buffer, display the result, and reset the buffer selection for the next operation.

Commands may be defined to operate on both single byte data or multi-byte data. In the code to follow, two commands have been defined, one for "Enter", and the other for "Add". These use the subroutines ENTDAT and ADDDAT respectively. The data routines store the hexadecimal equivalent of the data keys into the HEXDAT register and then shift this value into the selected buffer using the subroutine SHFDAT. The subroutine TBLLUT is used to look up the 16 bit address for a key routine.

The first 16 bit input buffer is defined by the two bytes BUFAH and BUFAL. Similarly, the second input buffer and the output buffer are defined by bytes BUFBH and BUFBL, and BUFCH and BUFCL. A control bit BUFSEL is used to select into which buffer the input data is shifted.

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

The following '8051 code reads the second table values and jump to a key routine where its code is executed:

```
KEYJMP:     MOV DPTR, #LUT2         ;load base address
            MOV A, LUT1OUT          ;load offset
            LCALL TBLLUT            ;get address from table 2 and load DPTR
            CLR A                   ;clear offset
            JMP @A+DPTR             ;jump to selected routine using value of DPTR

KNL:        ;NULL KEY ROUTINE
            LJMP IDLE               ;return to wait for next key

KC1:        ;COMMAND KEY F1 ROUTINE
            LCALL ENTDAT            ;code for F1 "Enter data" routine
            LJMP IDLE

KC2:        ;COMMAND KEY F2 ROUTINE
            LCALL ADDDAT            ;code for F2 "Add data" routine
            LJMP IDLE

KC3:        ;COMMAND KEY F3 ROUTINE
                                    ;code for F3 routine
            LJMP IDLE

KC4:        ;COMMAND KEY F4 ROUTINE
                                    ;code for F4 routine
            LJMP IDLE

KC5:        ;COMMAND KEY F5 ROUTINE
                                    ;code for F5 routine
            LJMP IDLE

KC6:        ;COMMAND KEY F6 ROUTINE
                                    ;code for F6 routine
            LJMP IDLE

KC7:        ;COMMAND KEY F7 ROUTINE
                                    ;code for F7 routine
            LJMP IDLE

KC8:        ;COMMAND KEY F8 ROUTINE
                                    ;code for F8 routine
            LJMP IDLE

KD0:        ;DATA KEY "0" ROUTINE
            MOV HEXDAT, #00H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key
```

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

```
KD1:        ;DATA KEY "1" ROUTINE
            MOV HEXDAT, #01H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD2:        ;DATA KEY "2" ROUTINE
            MOV HEXDAT, #02H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD3:        ;DATA KEY "3" ROUTINE
            MOV HEXDAT, #03H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD4:        ;DATA KEY "4" ROUTINE
            MOV HEXDAT, #04H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD5:        ;DATA KEY "5" ROUTINE
            MOV HEXDAT, #05H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD6:        ;DATA KEY "6" ROUTINE
            MOV HEXDAT, #06H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD7:        ;DATA KEY "7" ROUTINE
            MOV HEXDAT, #07H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD8:        ;DATA KEY "8" ROUTINE
            MOV HEXDAT, #08H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KD9:        ;DATA KEY "9" ROUTINE
            MOV HEXDAT, #09H        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KDA:        ;DATA KEY "A" ROUTINE
            MOV HEXDAT, #0AH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key
```

```
KDB:        ;DATA KEY "B" ROUTINE
            MOV HEXDAT, #0BH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KDC:        ;DATA KEY "C" ROUTINE
            MOV HEXDAT, #0CH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KDD:        ;DATA KEY "D" ROUTINE
            MOV HEXDAT, #0DH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KDE:        ;DATA KEY "E" ROUTINE
            MOV HEXDAT, #0EH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key

KDF:        ;DATA KEY "F" ROUTINE
            MOV HEXDAT, #0FH        ;convert the data to hexadecimal
            LCALL SHFDAT            ;shift data into selected buffer
            LJMP IDLE               ;return to wait for next key
```

This structure can easily be expanded to more keys as necessary.

The following are the subroutines called in the previous code. Note that the display portions are left as "an exercise for the reader" since the code will be specific to the display interface.

```
TBLLUT:     ;RETURN 16 BIT ADDRESS (256 WORD ENTRIES MAX)
            ;INPUTS FROM ACCUMULATOR AND DPTR
            ;OUTPUT TO DPTR
            CLR C
            RLC A               ;multiply A by two to select a two byte word
            ADDC A, DPL         ;add offset to low byte of base address
            MOV DPL, A
            MOV A, #00H
            ADDC A, DPH         ;add any carry to high byte of base address
            MOV DPH, A
            CLR A
            MOVC A, @A+DPTR     ;read msb 1st value to r0
            MOV R0, A
            INC DPTR
            CLR A
            MOVC A, @A+DPTR
            MOV DPL, A          ;read lsb 2nd value to dpl
            MOV DPH, R0         ;read msb value in r0 to dph
            RET
```

```
ENTDAT:     ;SELECTS BUFFER B
            SETB BUFSEL       ;select buffer B
            RET

ADDDAT:     ;ADDS NUMBERS IN BUFFERS A&B, RESULT IN BUFFER C
            MOV A, BUFAL
            ADD A, BUFBL      ;add low bytes in buffers A&B
            MOV BUFCL, A      ;store result in buffer C
            MOV A, BUFAH
            ADDC A, BUFBH     ;add high bytes in buffers A&B with carry
            MOV BUFCH, A      ;store result in buffer C
                              ;display carry & BUFC (exercise for reader!)
            CLR BUFSEL        ;select buffer A for next operation
            RET

SHFDAT:     ;SHIFTS DATA INTO ONE OF TWO BUFFERS
            JB BUFSEL, SHFDAT1      ;check which buffer to enter data into
            MOV A, BUFAH      ;shift data into BUFAH, BUFAL
            ANL A, #0FH       ;clear most significant nibble
            SWAP A            ;move least sig. nibble to most sig.nibble
            MOV BUFAH, A
            MOV A, BUFAL      ;read lower byte
            SWAP A            ;move most significant nibble to least position
            ANL A, #0FH       ;clear most significant position
            ORL A, BUFAH      ;combine most sig.BUFAL nibble with BUFAH
            MOV BUFAH, A      ;save updated higher byte
            MOV A, BUFAL      ;read lower byte
            ANL A, #0FH       ;clear most significant position
            SWAP A            ;move least sig.nibble to most sig.position
            ORL A, HEXDAT     ;combine hexdata into least sig.position
            MOV BUFAL, A      ;save updated lower byte
                              ;display buffer A (exercise for reader!)
            RET

      SHFDAT1:
            MOV A, BUFBH      ;shift data into BUFBH, BUFBL
            ANL A, #0FH       ;clear most significant nibble
            SWAP A            ;move least sig. nibble to most sig.nibble
            MOV BUFBH, A
            MOV A, BUFBL      ;read lower byte
            SWAP A            ;move most significant nibble to least position
            ANL A, #0FH       ;clear most significant position
            ORL A, BUFBH      ;combine most sig.BUFBL nibble with BUFBH
            MOV BUFBH, A      ;save updated higher byte
            MOV A, BUFBL      ;read lower byte
            ANL A, #0FH       ;clear most significant position
            SWAP A            ;move least sig.nibble to most sig.position
            ORL A, HEXDAT     ;combine hexdata into least sig.position
            MOV BUFBL, A      ;save updated lower byte
                              ;display buffer B (exercise for reader!)
            RET
```

Parsing Keyboard Commands and Data Using the E1115B

by Claude Haridge

In operation, the user types in the first hexadecimal number which is displayed as four hex values. After pressing the “Enter” key represented by F1, the user types in the second hexadecimal value. Pressing the “Add” key represented by F2 causes the two numbers to be added and displayed.

Parsing data and commands from the PS/2 keyboard converter's single byte output is therefore not too difficult and gives the developer the opportunity to implement a broad range of custom functionality.

References:

www.locus-engineering.com
E1115B PS/2 Keyboard to ASCII Converter Datasheet