13 Keyboard Interface

This chapter deals with the design of a keyboard interface that is implemented on UP2 using its FLEX 10K device. The chapter discusses how keyboards work and how they transmit data to and receive data from a computer. We will then take a simplified approach and show the interface for receiving data from a keyboard. The interface receives serial data from the keyboard and generates ASCII codes for keys that ASCII codes are applicable.

13.1 Data Transmission

Data communication between the keyboard and the host system is synchronous serial over bi-directional clock and data lines. Keyboard sends commands and key codes, and the system sends commands to the keyboard.

Either the system or the keyboard drive the data and clock lines, while clocking data in either direction is provided by the keyboard clock. When no communication is occurring, both lines are high. Figure 13.1 shows the timing of keyboard serial data transmission.

13.1.1 Serial Data Format

Data transmission on the data line is synchronized with the clock; data will be valid before the falling edge and after the rising edge of the clock pulse. Serial data transmission begins with the data line dropping to **0**. This bit value is taken on the rising edge of the clock and considered as the start-bit. On the next eight clock edges, data is transmitted in low to high order bit. The next data bit is the odd-parity bit, such that data bits and the parity bit always have odd number of ones. The last bit on the data line is the stop-bit that is always

1. After the stop-bit, the data line remains high until another transmission begins.

When the keyboard sends data to or receives data from the system it generates the clock signal to time the data. The system can prevent the keyboard from sending data by forcing the clock line to **0**, during this time the data line may be high or low. When the system sends data to the keyboard, it forces the data line to **0** until the keyboard starts to clock the data stream.

13.1.2 Keyboard Transmission

When the keyboard is ready to send data, it first checks the status of the clock to see if it is allowed to transmit data. If the clock line is forced to low by the system, data transmission to the system is inhibited and keyboard data is stored in the keyboard buffer. If the clock line is high and the data line is low, the keyboard is to receive data from the system. In this case, keyboard data is stored in the keyboard buffer, and the keyboard receives system data. If the clock and data lines are both high the keyboard sends the start-bit, 8 data bits, the parity bit and the stop-bit.

During transmission, the keyboard checks the clock line for low level at least every 60 μseconds. If the system forces the clock line to **0** after the keyboard starts ending data, a condition known as line contention occurs, and the keyboard stops sending data. If line contention occurs before the rising edge of the 10th clock pulse, the keyboard buffer returns the clock and data lines to high level.

13.1.3 System Transmission

The system sends 8-bit commands to the keyboard. When the system is ready to send a command to the keyboard, it first checks to see if the keyboard is sending data. If the keyboard is sending, but has not reached the 10th clock signal, the system can override the keyboard output by forcing the keyboard clock line to **0**. If the keyboard transmission is beyond the 10th clock signal, the system receives the transmission.

If the keyboard is not sending or if the system decides to override the output of the keyboard, the system forces the keyboard clock line to **0** for more than 60 μseconds while preparing to send data. When the system is ready to send the start bit, it allows the keyboard to drive the clock line to **1** and drives the data line to low. This signals the keyboard that data is being transmitted from the system. The keyboard generates the clock signals and receives the data bits, parity and the stop-bit. After the stop-bit, the system releases the data line. If the keyboard receives the stop-bit it forces the data line low to signal the system that the keyboard has received its data.

Upon receipt of this signal, the system returns to a ready state, in which it can accept keyboard output or goes to the inhibited state until it is ready. If the keyboard does not receive the stop-bit, a framing error has occurred, and the keyboard continues to generate clock signals until the data line becomes high. The keyboard then makes the data line low and requests a resending of the data. A parity error will also generate a re-send request by the keyboard.

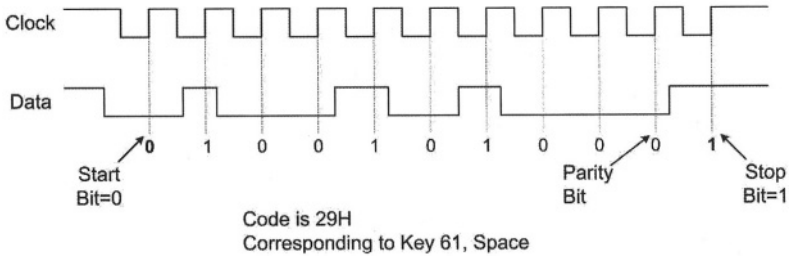

Figure 13.1 Keyboard Serial Data Transmission

13.1.4 Power-On Routine

The keyboard logic generates a power-on-reset signal when power is first applied to the keyboard. The timing of this signaling is between 150 milliseconds and 2.0 seconds from the time power is first applied to the keyboard.

Following this signaling, basic assurance test is performed by the keyboard. This test consists of a keyboard processor test, a checksum of its ROM, and a RAM test. During this test, activities on the clock and data lines are ignored. The keyboard LEDs are turned on at the beginning and off at the end of the test. This test takes a minimum of 300 milliseconds and a maximum of 500 milliseconds. Upon satisfactory completion of the basic assurance test, a completion code (hex AA) is sent to the system, and keyboard scanning begins.

Table 13.1 System Commands to Keyboard

Command	Hex
Set/Reset Status Indicators	ED
Echo	EE
Invalid Command	EF
Select Alternate Scan Codes	F0
Invalid Command	F1
Read ID	F2
Set Typematic Rate/Delay	F3
Enable	F4
Default Disable	F5
Set Default	F6
Set All Keys – Typematic - Make/Break - Make - Typematic/Make/Break	F7 F8 F9 FA
Set Key Type – Typematic - Make/Break - Make	FB FC FD
Resend	FE
Reset	FF

13.2 Codes and Commands

A host system may send 8-bit commands to the keyboard, while a keyboard may send commands and key codes to the system.

13.2.1 System Commands

System commands may be sent to the keyboard at any time. The keyboard will respond within 20 milliseconds, except when performing the basic assurance test (BAT), or executing a Reset command. System commands and their hexadecimal values are shown in Table 13.1.

13.2.2 Keyboard Commands

Table 13.2 shows the commands that the keyboard may send to the system and their hexadecimal values.

Table 13.2 Keyboard Commands to System

Command	Hex
Key Detection Error/Overrun	00
Keyboard ID	83AB
BAT Completion Code	AA
BAT Failure Code	FC
Echo	EE
Acknowledge (ACK)	FA
Resent	FE

13.2.3 Keyboard Codes

Keyboards are available for several languages and settings. The keyboard that is most common for the English language is one with 104 keys shown in Figure 13.2. Keys of this keyboard are identified by numbers, and for every key there is a scan code. Several scan codes are available, and the default scan code is Scan Code 2 that we will discuss here.

Keyboard scan codes consist of a Make and a Break code. The Make code identifies the key pressed and the Break code indicates the release of a key. For most keys the Break code is F0 followed by the Make code. For example when the Space bar (key 61) is pressed and released, hexadecimal codes 29, F0 and 29 are transmitted from the keyboard to the system via the data serial line. If this key remains pressed, the Make code (29) is continuously transmitted until it is released. Make codes for Scan Code 2 are shown in Table 13.3

The Make and Break arrangement, makes it possible for the system to identify multiple keys pressed and the order in which they have been pressed. For example, if one presses and holds down the Left-Shift key (key number 44), 12 Hex is continuously sent to the system. While this is happening, if key number 9 (the 8/* key) is pressed and released, 3E, F0 and 3E codes are transmitted. The receiving system identifies this sequence of events as the intention to enter an asterisk (*).

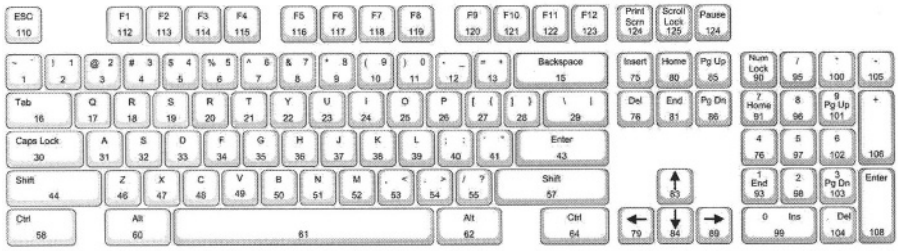

Figure 13.2 Standard 104-key Keyboard and Key Numbers

Table 13.3 Keyboard Scan Codes and Corresponding ASCII Characters

Key Numb	Make Code	ASCII		Character	
		No Shift	Shift	No Shift	Shift
1	0E	96	126		~
2	16	49	33	1	!
3	1E	50	64	2	@
4	26	51	35	3	#
5	25	52	36	4	$
6	2E	53	37	5	%
7	36	54	94	6	^
8	3D	55	38	7	&
9	3E	56	42	8	*
10	46	57	40	9	(
11	45	48	41	0	)
12	4E	45	95	-	_
13	55	61	43	=	+
15	66	08	08	BS	BS
16	0D	09	09	Tab	Tab
17	15	81	113	Q	q
18	1D	87	119	W	w
19	24	69	101	E	e
20	2D	82	114	R	r
21	2C	84	116	T	t
22	35	89	121	Y	y
23	3C	85	117	U	u
24	43	73	105	I	i
25	44	79	111	O	o
26	4D	80	112	P	p
27	54	91	123	[	{
28	5B	93	125	]	}
29	5D	92	124	\	|
30	58			Caps	Caps
31	1C	65	97	A	a
32	1B	83	115	S	s
33	23	68	100	D	d

Key Numb	Make Code	ASCII		Character	
		No Shift	Shift	No Shift	Shift
34	2B	70	102	F	f
35	34	71	103	G	g
36	33	72	104	H	h
37	3B	74	106	J	j
38	42	75	107	K	k
39	4B	76	108	L	l
40	4C	59	58	;	:
41	52	39	34	'	"
42	5D				
43	5A	13	Enter	Enter	Enter
44	12		Shift	Shift	Shift
45	61				
46	1A	90	122	Z	z
47	22	88	120	X	x
48	21	67	99	C	c
49	2A	86	118	V	v
50	32	66	98	B	b
51	31	78	110	N	n
52	3A	77	109	M	m
53	41	44	60	,	<
54	49	46	62	.	>
55	4A	47	63	/	?
57	59			Shift	Shift
58	14			Ctrl	Ctrl
59	E0 1F			Win	Win
60	11			Alt	Alt
61	29	32	32	Space	Space
62	E0/11			Alt	Alt
63	E0 27			Win	Win
64	E0/14			Ctrl	Ctrl
65	E0 2F			Menu	Menu

13.3 Keyboard Interface Design

This section discusses a keyboard interface for reading scan data from the keyboard and producing ASCII codes of the keys pressed. Code Set 2 is assumed, and the interface only handles data transmission from the keyboard. The interface reads serial data from the keyboard, detects the Make code when a key is pressed and looks up the Make code in an ASCII conversion table. For simplicity, the look-up table only handles upper-case characters.

13.3.1 Collecting the Make Code

The first part of our interface connects to the keyboard data and clock lines and when a key is pressed, it outputs an 8-bit scan code. The *KBdata, KBclock* inputs are for the keyboard data and clock inputs, and the 8-bit *ScanCode* is the main output of this part.

This part also uses a fast synchronizing clock, *SYNclk,* and a keyboard reset input, *KBreset.* In addition to the *ScanCode* output, this part outputs a signal to indicate that a scan code is ready (*ScanRdy*) and another output to indicate that a key has been released (*KeyReleased*). These outputs make distinction between Make and Break states.

```
module KeyboardInterface
  (KBclk, KBdata, ResetKB, SYNclk, ScanRdy, ScanCode, KeyReleased);
  input KBclk;
  input KBdata;
  input ResetKB;
  input ReadKB;
  input SYNclk;
  output ScanRdy;
  output [7:0] ScanCode;
  output KeyReleased;

  // Details in Figure 13.4
  // Generate an internal synchronized clock
  always @(posedge Clock) begin
    . . .
    // Count the number of serial bits and collect data into ScanCode
    . . .
  end

  // Details in Figure 13.5
  always @(posedge SYNclk) begin
    . . .
    // Keep track of the state of Scan Codes outputted
    . . .
  end
  // Issue KeyReleased when done

endmodule
```

Figure 13.3 Verilog Pseudo Code

The pseudo-code of this unit is shown in Figure 13.3. After the declarations, in this part an internal clock (*Clock*) that is based on the keyboard clock and is synchronized with the board clock is generated. This clock is used in an **always** block to collect serial data bits and shift them into *ScanCode*. Another **always** block in this code, monitors completion of serial data collection and generates the state of the keys pressed and released. The details of these sections of the interface module are depicted in Figure 13.4 and Figure 13.5 respectively.

The first **always** statement of Figure 13.4 shows the generation of *Clock* that is equal to the keyboard clock and synchronized with the external system clock, *SYNclk*. In the **always** block that follows this block, after detection of the start-bit, on the rising edge of *Clock*, bit values are read from the keyboard data line (*KBdata*) and shifted into *ScanCode*. The shifting continues for 8 bit counts. On the next clock after collecting 8 data bits is complete, *ScanRdy is* issued, and the collection process returns to its initial state of looking for the next start-bit.

```
. . .
reg Clock;

always @ (posedge SYNclk) Clock = KBclk;

reg [3:0] BitCount;
reg StartBitDetected, ScanRdy;
reg [7:0] ScanCode;

always @(posedge Clock) begin
  if (ResetKB) begin
    BitCount = 0; StartBitDetected = 0;
  end else begin
    if (KBdata == 0 && StartBitDetected == 0) begin
      StartBitDetected = 1;
      ScanRdy = 0;
    end else if (StartBitDetected) begin
      if (BitCount < 8) begin
        BitCount = BitCount + 1;
        ScanCode = {KBdata, ScanCode[7:1]};
      end else begin
        StartBitDetected = 0;
        BitCount = 0;
        ScanRdy =1;
      end
    end
  end
end
. . .
```

Figure 13.4 Serial Data Collection

The partial code of Figure 13.5 uses the two-bit *CompletionState* to keep track of the scan codes that have been generated. Starting in the initial state,

when *ScanRdy* becomes **1** and F0 is on *ScanCode, CompletionState* becomes 1. This state is entered when a key is released and the F0 part of the Break code is transmitted. The next time *ScanRdy* is detected, the second part of the Break code (that is the same as Make) becomes available on *ScanCode*. In the following clock, the *KeyReleased* output becomes **1** and remains at this level for a complete clock period.

```
  . . .
  reg [1:0] CompletionState;
  wire KeyReleased;

  always @ (posedge SYNclk) begin
    if (ResetKB) CompletionState = 0;

    else case (CompletionState)
      0: if (ScanCode == 8'hF0 && ScanRdy == 1) CompletionState = 1;
        else CompletionState = 0;
      1: if (ScanRdy == 1) CompletionState = 1;
        else CompletionState = 2;
      2: if (ScanRdy == 0) CompletionState = 2;
        else CompletionState = 3;
      3: CompletionState = 0;
    endcase

  end

  assign KeyReleased = CompletionState == 3 ? 1 : 0;

endmodule
```

Figure 13.5 Break State Recognition

13.3.2 ASCII Look-Up

The ASCII lookup part of our keyboard interface is a ROM of Quartus II megafunctions with 7 address lines and word length of 8 bits. Hexadecimal locations 0D through 66 of this ROM are defined to contain ASCII codes for scan codes that correspond to ROM addresses. This megafunction is defined to use the *KbASCII.mif* memory initialization file, a portion of which is shown in Figure 13.6. The *ScanCode* output of Figure 13.3 connects to the address input of this ROM, and ASCII codes corresponding to input addresses appear on its output.

```
DEPTH = 128;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = DEC;
% Keyboard Scan Code to ASCII %
CONTENT
BEGIN
% Set 2:   ASCII    ;        Key   Char    %
%------+--------------+-----------------%
0D     :      09     ;  %     16     Tab   %
0E     :      96     ;  %     1             %
11     :      0      ;  %     60     Alt   %
12     :      0      ;  %     44     Shift %
14     :      0      ;  %     58     Ctrl  %
15     :      81     ;  %     17     Q     %
16     :      49     ;  %     2      1     %
1A     :      90     ;  %     46     Z     %
1B     :      83     ;  %     32     S     %
1C     :      65     ;  %     31     A     %
. . .
66     :      08     ;  %     15     BS    %
END;
```

Figure 13.6 ASCII Conversion Memory Initialization File

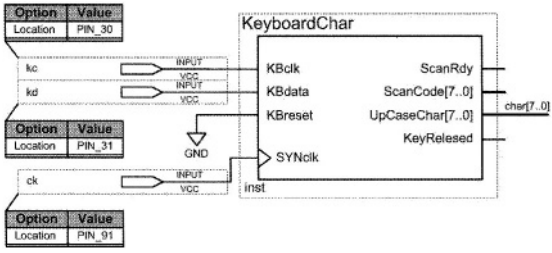

Figure 13.7 Prototyping Keyboard Character Generator

13.4 Keyboard Interface Prototyping

The Verilog module of Figure 13.3 and the ROM of Figure 13.6 are put together into the *KeyboardChar* schematic file. Testing this design is achieved by programming the FLEX 10K of UP2, assigning keyboard clock and data inputs to pins 30 and 31 (see Figure 6.37), and connecting the UP2 clock to its *SYNclk* input. A portion of the schematic of this prototype design is shown in Figure 13.7. With this settings, the ASCII code of the key pressed on the keyboard that is connected to the PS2 connecter of UP2 appears on the 8-bit char output of the diagram of Figure 13.7.

13.5 Summary

This chapter showed another interface design utilizing Verilog design entry as well as storage megafunctions of Quartus II. The design here illustrated how an interface could be designed to read keyboard characters. The knowledge of this basic peripheral is important for logic designers and students in this field.