Designing a USB Keyboard with the Cypress Semiconductor CY7C63413 USB Microcontroller

Introduction

The Universal Serial Bus (USB) is an industry standard serial interface between a computer and its peripherals such as a keyboard, mouse, joystick, etc. This application note describes how to design a USB keyboard using the Cypress Semiconductor single-chip CY7C63413 USB microcontroller. The document starts with the basic operations of a PS/2 keyboard followed by an introduction to the CY7C63413 USB controller. A schematic of the USB keyboard and its connection details can be found in the Hardware Implementation Section.

The software section of this application note describes the architecture of the firmware required to implement the keyboard function. Several flowcharts are included to assist in the explanation. The source code of the demonstration keyboard firmware is available with the Cypress CY3651 Development Kit. Please contact your local Cypress Sales Office for details.

This application note assumes that the reader is familiar with the CY7C63413 USB controller and the Universal Serial Bus. The CY7C63413 data sheet is available from the Cypress web site at http://www.cypress.com. USB documentation can be found at the USB Implementers Forum web site at http://www.usb.org/.

PS/2 Keyboard Basics

Key Switches and Scan Matrix

A PS/2 keyboard typically has between 101 and 104 keys that are uniquely positioned in a scan matrix. The scan matrix consists of M rows and N columns, all of which are electrically isolated from each other. On average, the number of rows (M) is no greater than 8, and the number of columns (N) is no greater than 20. Each key sits over two isolated contacts of its corresponding row and column in the scan matrix. When a key is pressed, the two contacts are shorted together, and the row and column of the key are electrically connected (*Figure 1*).

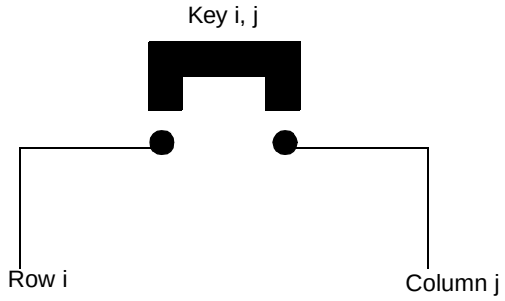

Figure 1. Key Switch

PS/2 Controller

The PS/2 keyboard contains an embedded controller that performs a variety of tasks, all of which help to cut down on the overall system overhead. The essential task of the PS/2 controller is to monitor the keys and report to the main computer whenever a key is pressed or released. The controller writes a scan pattern out to the column lines consisting of all 1s and one 0 which is shifted through each column. The result is then read at the row lines. If a 0 is propagated to a row line, then the key at the intersection of that column and row has been pressed. See *Figure 2* and *Figure 3*.

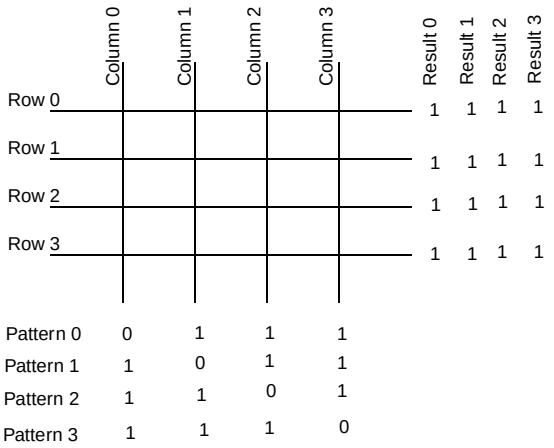

Figure 2. Scan Results For No Key Press

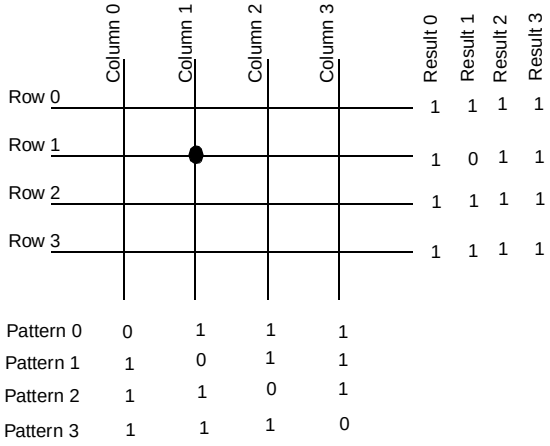

Figure 3. Scan Results for Key $_{1,1}$ Pressed

PS/2 Cable

The PS/2 keyboard is connected to the main computer through a shielded PS/2 cable that usually contains 6 wires carrying 4 signals: Vcc, GND, Clock, and Data. The remaining 2 wires are unused (one of them is sometimes tied to chassis ground.)

Introduction to CY7C63413

The CY7C63413 is a high performance 8-bit RISC microcontroller with an integrated USB Serial Interface Engine (SIE). The architecture implements 37 commands that are optimized for USB applications. The CY7C63413 has a built-in clock oscillator and timers, as well as general purpose I/O lines that can be configured as resistive with internal pull-ups, open-drain outputs, or traditional CMOS outputs. High performance, low-cost human-interface type computer peripherals such as a keypad or keyboard can be implemented with minimum external components and firmware effort.

Clock Circuit

The CY7C63413 has a built-in clock oscillator and PLL-based frequency doubler. This circuit allows a cost effective 6 MHz ceramic resonator to be used externally while the on-chip RISC core runs at 12 MHz.

USB Serial Interface Engine (SIE)

The operation of the SIE is totally transparent to the user. In receive mode, USB packet decode and data transfer to the endpoint FIFO are automatically done by the SIE. The SIE then generates an interrupt to invoke the service routine after a packet is unpacked.

In the transmit mode, data transfer from the endpoint and the assembly of the USB packet are handled automatically by the SIE.

General Purpose I/O

The CY7C63413 has 32 general purpose I/O lines divided into 4 ports: Port 0 through Port 3. One such I/O circuit is shown in *Figure 4*. Each port (8 bits) can be configured as resistive with internal pull-ups (7 KΩ), open drain outputs (high impedance inputs), or traditional CMOS outputs.The port configuration is determined according to *Table 1* below**.**

Table 1. GPIO Configuration

Port Configuration Bits	Pin Interrupt Bit	Driver Mode	Interrupt Polarity
11	X	Resistive	-
10	0	CMOS Output	disabled
10	1	CMOS Input	disabled
01	X	Open Drain	-
00	X	Open Drain	+

Ports 0 to 2 offer low current drive with a typical current sink capability of 7 mA. Port 3 offers higher current drive, with a typical current sink of 12 mA which can be used to drive LEDs.

Each General Purpose I/O (GPIO) is capable of generating an interrupt to the RISC core. Interrupt polarity is selectable on a per port basis using the GPIO Configuration Register (see *Table 1* above.) Selecting a negative polarity (“-”) will cause falling edges to trigger an interrupt, while a positive polarity (“+”) selects rising edges as triggers. The interrupt triggered by a GPIO line is individually enabled by a dedicated bit in the Interrupt Enable Register. All GPIO interrupts are further masked by the Global GPIO Interrupt Enable Bit in the Global Interrupt Enable Register.

The GPIO Configuration Register is located at I/O address 0x08. The Data Registers are located at I/O addresses 0x00 to 0x03 for Port 0 to Port 3 respectively.

Power-up Mode

The CY7C63413 offers 2 modes of operation after a power-on-reset (POR) event: suspend-on-reset (typical for a USB application) and run-on-reset (typical for a PS/2 application). The suspend-on-reset mode is selected by attaching a pull-up resistor (100K to 470K Ω) to Vcc on Bit 7 of GPIO Port 3. The run-on-reset mode is selected by attaching a pull-down resistor (0 to 470K Ω) to ground on Bit 7 of GPIO Port 3. See *Figure 5* and *Figure 6*.

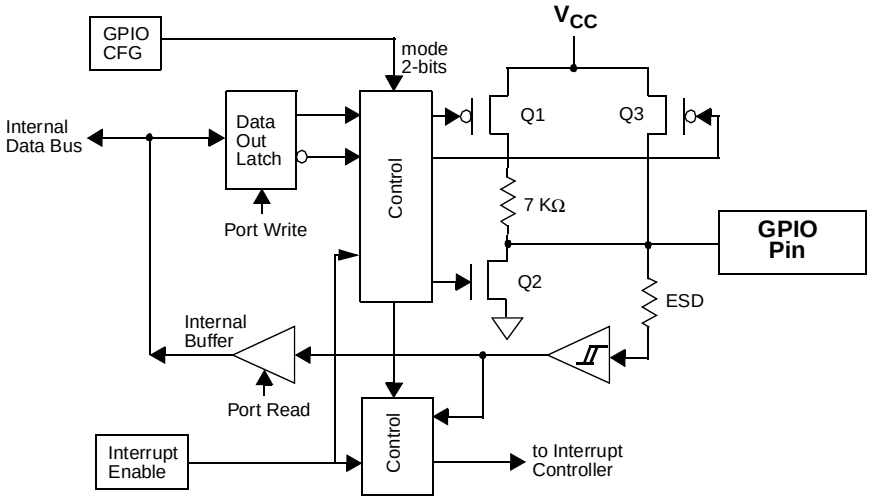

Figure 4. One General Purpose I/O Line

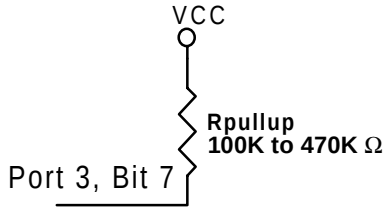

Figure 5. Suspend-On-Reset Mode

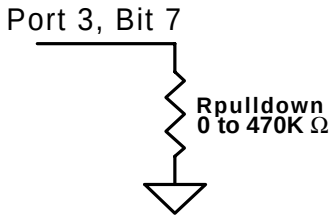

Figure 6. Run-On-Reset Mode

Hardware Implementation

Figure 8 is the schematic for a keyboard application.

Port 2 is configured as resistive (7 Kohm pull-ups to Vcc), and is connected to the M rows of the scan matrix (up to 8 rows are supported). Port 0, 1, and 3 are also configured as resistive, and are connected to the N columns of the scan matrix (up to 20 columns are supported.) The 3 LEDs (Num Lock, Caps Lock, and Scroll Lock) are connected to the lower three bits of Port 3 as well (for high current drive.) For scan matrices with less than 8 rows or 20 columns, the unused port bits should be left unconnected (internally, they are pulled up).

During a scan test where no key is pressed, the row port data bits will be high since all row lines are internally pulled up to Vcc. When a key is pressed, setting its column port line low will cause its row line to go low as well. See *Figure 7*.

Port 3, Bit 7 is pulled up to Vcc through a 470K Ω resistor (R5) so that the microcontroller will suspend after a POR. The microcontroller will resume once it detects bus activity (i.e USB Bus Reset).

A 6 MHz ceramic resonator is connected to the clock inputs of the microcontroller. This component should be placed as close to the microcontroller as possible.

Since the keyboard is a bus-powered device, the 5V Vcc and GND come directly from the USB cable. The Vcc pin is connected to the Vcc of the cable through a ramp regulating resistor of 1.5 Ω (R6). This resistor, along with a 22 µF capacitor (C2) connected between the Vcc and GND pins, provide roughly a 30 µs power ramp time. Vcc is also bypassed for high frequency noise by a 0.1 µF capacitor (C1).

According to the USB specification, the USB D– line of a low-speed device (1.5 Mbps) should be tied to a voltage source between 3.0V and 3.6V with a 1.5 KΩ pull-up terminator. The CY7C63413 eliminates the need for a 3.3V regulator by specifying a 7.5 KΩ resistor (R1) connected between the USB D– line and the nominal 5V Vcc.

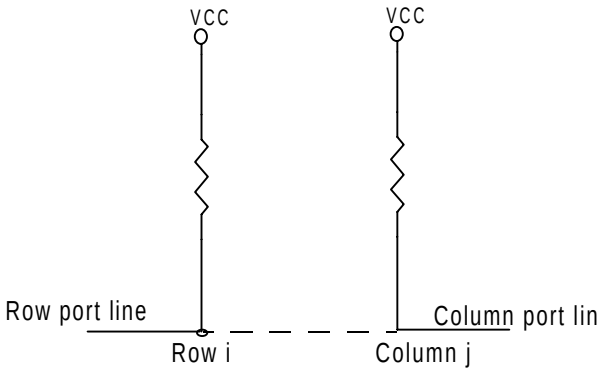

Figure 7. Row/Column Port Configuration

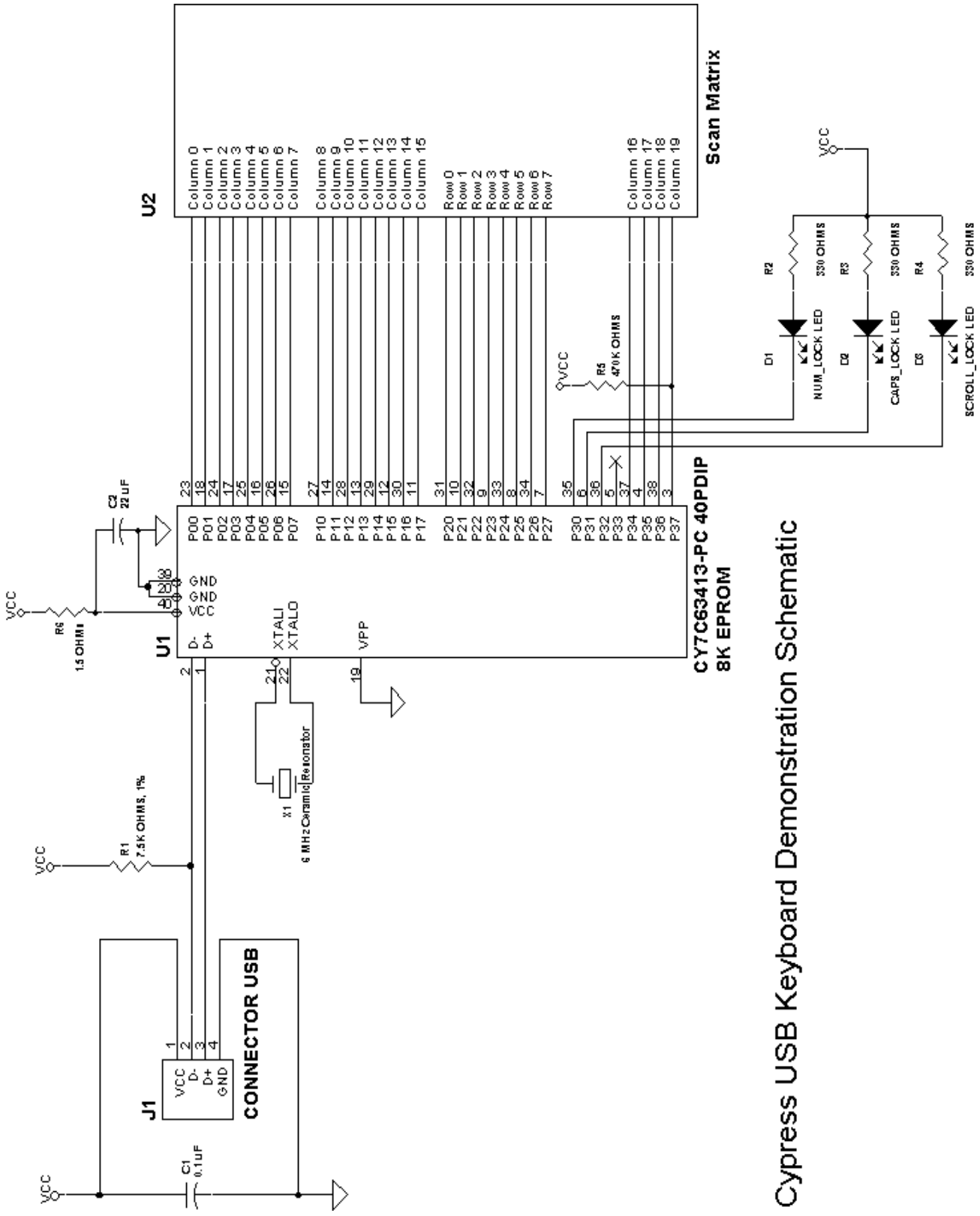

Figure 8. Hardware Implementation

Firmware Implementation

USB Interface

All USB Human Interface Device (HID) class applications such as a keyboard, follow the same USB start-up procedure. The procedure is as follows (see *Figure 9*.)

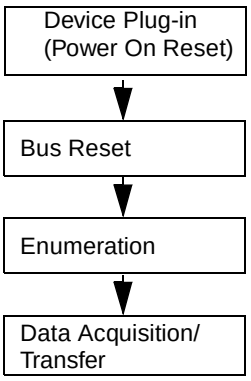

Figure 9. USB Start-Up Procedure

Device Plug-in (Power On Reset)

The USB device is powered when it is connected to the bus. The pull-up resistor on D– notifies the hub that a low-speed (1.5 Mbps) device has just been connected. Program execution begins at address 0 (see *Figure 10*).

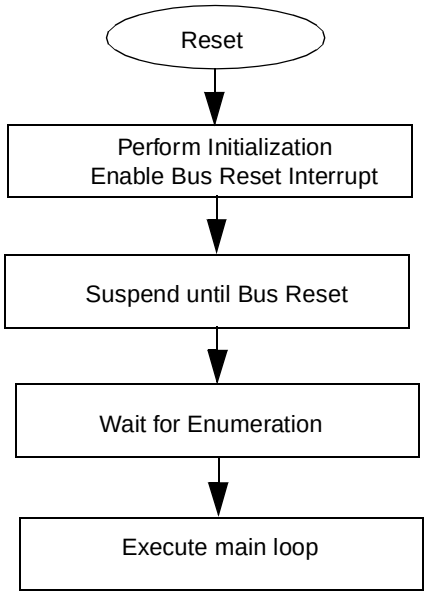

Figure 10. Power On Reset

Bus Reset

The host recognizes the presence of a new USB device and resets it (see *Figure 11*).

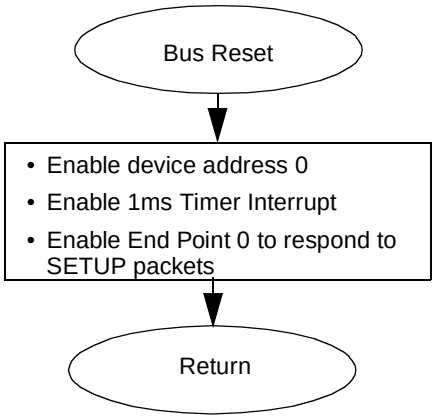

Figure 11. Bus Reset ISR

Enumeration

The host sends a SETUP packet followed by IN packets to read the device description from default address 0. When the description is received, the host assigns a new USB address to the device. The device begins responding to communication with the newly assigned address. The host then asks for the device descriptor, configuration descriptor and HID report descriptor. The descriptors hold the information about the device. They will be discussed in detail below. Using the information returned from the device, the host now knows the number of data endpoints supported by the device (in a USB keyboard, there is only one data endpoint). The host will issue a Set Configuration with a value of 1. At this point, the process of enumeration is completed. See *Figures 12*, *13,* and *14*.

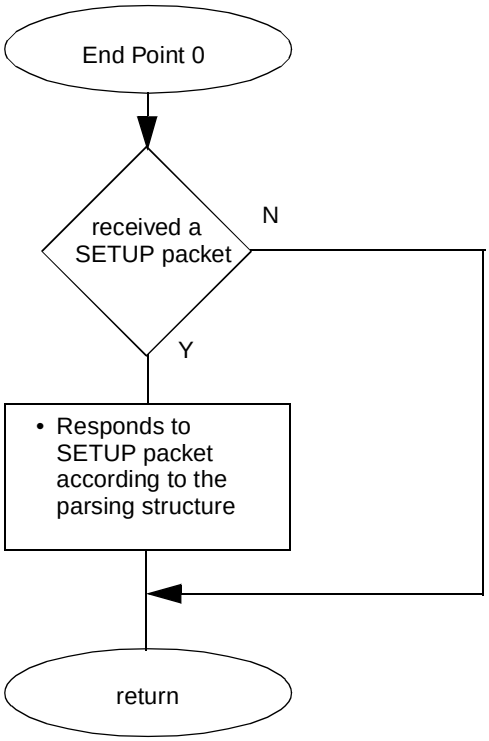

Figure 12. Endpoint 0 ISR

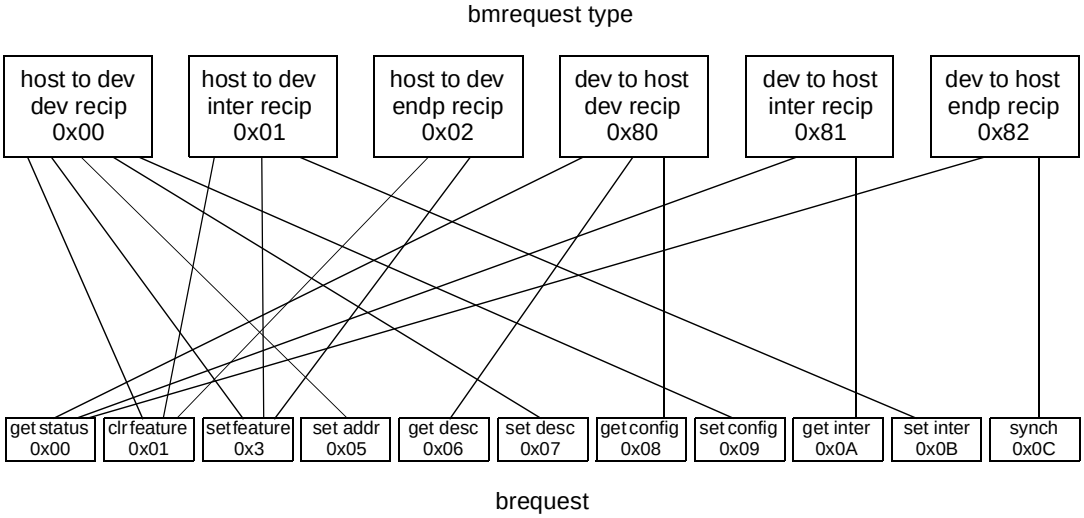

Figure 13. USB Standard Request Parsing Structure

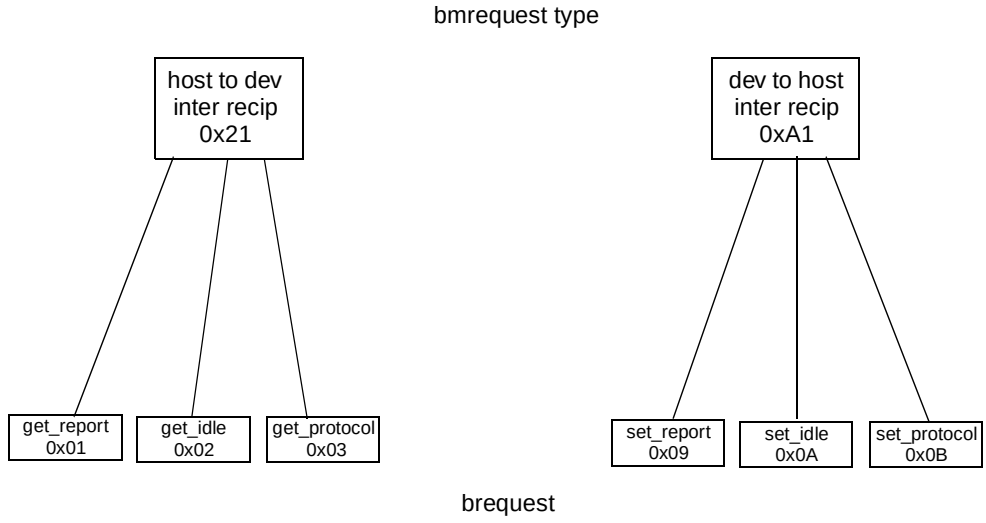

Figure 14. USB HID Class Request Parsing Structure

Data Acquisition/Transfer

The firmware periodically writes scan patterns to the scan matrix columns, and reads the row result to determine which keys are pressed. The scan codes of the keys pressed are sent to the host using endpoint 1 (see *Figure 15*). When the host issues IN packets to retrieve data from the device, the device returns eight bytes of data. These eight bytes hold the keyboard control data (see *Figure 16*). Usage codes for each key can be found in Appendix A.3 of the Device Class Definition for Human Interface Devices (HID). When one of the LED buttons (i.e. Num Lock, Caps Lock, Scroll Lock) are pressed or released, the host issues a SETUP packet with a Set_Report request through the control pipe to End Point 0, followed by an OUT packet with 1 Data byte indicating which LED should be on or off (see *Figure 17*.)

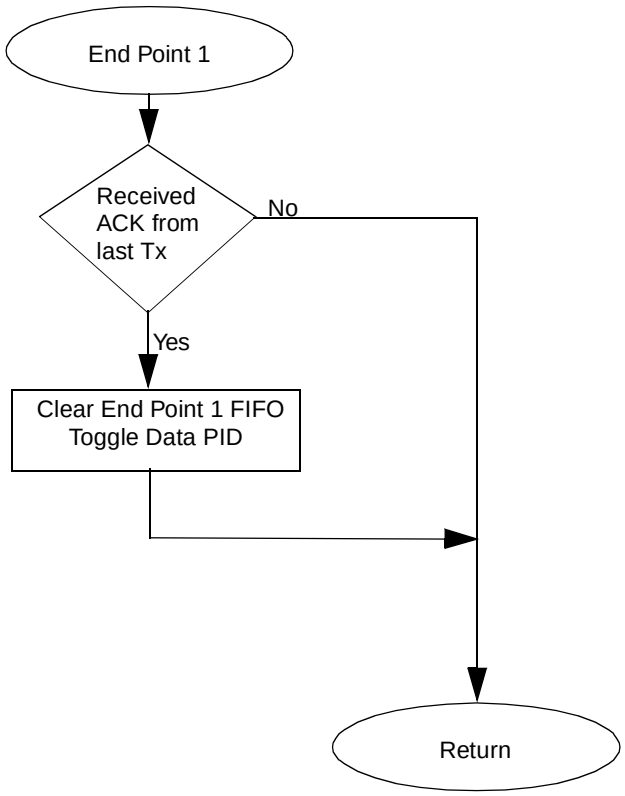

Figure 15. Endpoint 1 Interrupt Service Routine

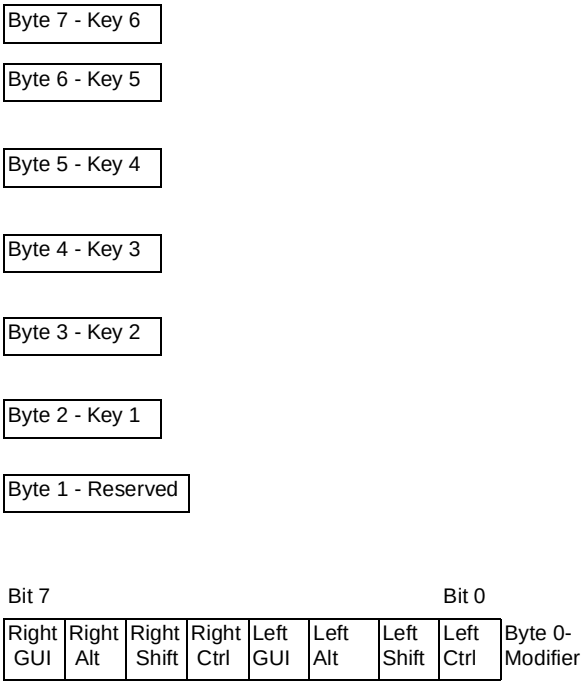

Figure 16. IN Data Organization for USB Keyboard

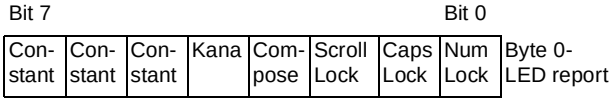

Figure 17. LED Report for USB Keyboard

The byte order and bit field positions are defined by the HID report descriptor (discussed below), and are also consistent with the *Boot Protocol* requirements for legacy systems.

USB Descriptors

As stated earlier, the USB descriptors hold information about the device. There are several types of descriptors, which will be discussed in detail below. All descriptors have certain characteristics in common. Byte 0 is always the descriptor length in bytes and Byte 1 is always the descriptor type. Discussion of these two bytes will be omitted from the following descriptions. The rest of the descriptor structure is dependent on the descriptor type. An example of each descriptor will be given. Descriptor types are device, configuration, interface, endpoint, string, report, and several different class descriptors.

Device Descriptor

This is the first descriptor the host requests from the device. It contains important information about the device. The size of this descriptor is 18 bytes. A list follows:

- USB Specification release number in binary-coded decimal (BCD) (2 bytes)
- Device class (1 byte)
- Device subclass (1 byte)
- Device protocol (1 byte)
- Max packet size for Endpoint 0 (1 byte)
- Vendor ID (2 bytes)
- Product ID (2 bytes)
- Device release number in BCD (2 bytes)
- Index of string describing Manufacturer (Optional) (1 byte)
- Index of string describing Product (Optional) (1 byte)
- Index of string containing serial number (Optional) (1 byte)
- Number of configurations for the device (1 byte)

```
Example of a device descriptor

Descriptor Length (18 bytes)
Descriptor Type (Device)
Complies to USB Spec Release (1.00)
Class Code (insert code)
Subclass Code (0)
Protocol (No specific protocol)
Max Packet Size for endpt 0 (8 bytes)
Vendor ID (Cypress)
Product ID (USB Keyboard)
Device Release Number (1.03)
String Describing Vendor (None)
String Describing Product (None)
String for Serial Number (None)
Possible Configurations (1)
```

Configuration Descriptor

The configuration descriptor is 9 bytes in length and gives the configuration information for the device. It is possible to have more than one configuration for each device. When the host requests a configuration descriptor, it will continue to read these descriptors until all configurations have been received. A list of the structure follows:

- Total length of the data returned for this configuration (2 bytes)
- Number of interfaces for this configuration (1 byte)
- Value used to address this configuration (1 byte)
- Index of string describing this configuration (Optional) (1 byte)
- Attributes bitmap describing configuration characteristics (1 byte)
- Maximum power the device will consume from the bus (1 byte)

```
Example of configuration descriptor

Descriptor Length (9 bytes)
Descriptor Type (Configuration)
Total Data Length (34 bytes)
Interfaces Supported (1)
Configuration Value (1)
String Describing this Config (None)
Config Attributes (Bus powered)
Max Bus Power Consumption (100mA)
```

Interface Descriptor

The interface descriptor is 9 bytes long and describes the interface of each device. It is possible to have more than one interface for each device. This descriptor is set up as follows:

- Number of this interface (1 byte)
- Value used to select alternate setting for this interface (1 byte)
- Number of endpoints used by this interface. If this number is zero, only endpoint 0 is used by this interface (1 byte)
- Class code (1 byte)
- Subclass code (1 byte)
- Protocol code (1 byte)
- Index of string describing this interface (1 byte)

```
Example of interface descriptor

Descriptor Length (9 bytes)
Descriptor Type (Interface)
Interface Number (0)
Alternate Setting (0)
Number of Endpoints (1)
Class Code (insert code)
Subclass Code (0)
Protocol (No specific protocol)
String Describing Interface (None)
```

HID (Class) Descriptor

The class descriptor tells the host about the class of the device. In this case, the device falls in the human interface device (HID) class. This descriptor is 9 bytes in length and is set up as follows:

- Class release number in BCD (2 bytes)
- Localized country code (1 byte)
- Number of HID class descriptor to follow (1 byte)
- Report descriptor type (1 byte)
- Total length of report descriptor in bytes (2 bytes)

```
Example of HID class descriptor

Descriptor Length (9 bytes)
Descriptor Type (HID Class)
HID Class Release Number (1.00)
Localized Country Code (USA)
Number of Descriptors (1)
Report Descriptor Type (HID)
Report Descriptor Length (63 bytes)
```

Endpoint Descriptor

The endpoint descriptor describes each endpoint, including the attributes and the address of each endpoint. It is possible to have more than one endpoint for each interface. This descriptor is 7 bytes long and is set up as follows:

- Endpoint address (1 byte)
- Endpoint attributes. Describes transfer type (1 byte)
- Maximum packet size this endpoint is capable of transferring (2 bytes)
- Time interval at which this endpoint will be polled for data (1 byte)

```
Example of endpoint descriptor

Descriptor Length (7 bytes)
Descriptor Type (Endpoint)
Endpoint Address (IN, Endpoint 1)
Attributes (Interrupt)
Maximum Packet Size (8 bytes)
Polling Interval (10 ms)
```

Report Descriptor

This is the most complicated descriptor in USB. There is no set structure. It is more like a computer language that describes the format of the device's data in detail. This descriptor is used to define the structure of the data returned to the host as well as to tell the host what to do with that data. An example of a report descriptor can be found below.

A report descriptor must contain the following items: Input (or Output or Feature), Usage, Usage Page, Logical Minimum, Logical Maximum, Report size, and Report Count. These are all necessary to describe the device's data.

```
Example of report descriptor

Usage Page (Generic Desktop)
Usage (Keyboard)
Collection (Application)
     Usage Page(key codes)
     Usage Minimum (224)
     Usage Maximum (231)
     Logical Minimum (0)
     Logical Maximum (1)
     Report Size (1)
     Report Count (8) ; modifier byte
     Input (Data, Variable, Absolute)

     Report Count (1)
     Report Size (8)
     Input (Constant) ; reserved byte
```

```
        Report Count (5)
        Report Size (1)
        Usage Page (LEDs)
        Usage Minimum (1)
        Usage Maximum (5)
        Output (Data, Variable,
        Absolute); LED report
        Report Count (1)
        Report Size (3)
        Output (Constant) ;padding

        Report Count (6)
        Report Size (8)
        Logical Minimum (0)
        Logical Maximum (101)
        Usage Page (key codes)
        Usage Minimum (0)
        Usage Maximum (101)
        Input (Data, Array) ;key array(6)
    End Collection
```

Input items are used to tell the host what type of data will be returned as input to the host for interpretation. These items describe attributes such as data vs. constant, variable vs. array, absolute vs. relative, etc.

Usages are the part of the descriptor that defines what should be done with the data that is returned to the host. From the example descriptor, Usage (Keyboard) tells the host that this is a keyboard device. There is also another kind of Usage tag found in the example called a Usage Page. The reason for the Usage Page is that it is necessary to allow for more than 256 possible Usage tags. Usage Page tags are used as a second byte which allows for up to 65536 Usages.

Logical Minimum and Logical Maximum are used to bound the values that a device will return. For example, a keyboard that will return the values 0 to 101 for the scan code of each key press will have a Logical Minimum (0) and Logical Maximum (101). These are different from Physical Minimum and Physical Maximum. Physical boundaries give some meaning to the Logical boundaries. For example, a thermometer may have Logical boundaries of 0 to 999, but the Physical boundaries may be 32 to 212. In other words, the boundaries on the thermometer are 32 to 212 degrees Fahrenheit, but there are one thousand steps defined between the boundaries.

Report Size and Report Count define the structures that the data will be transferred in. Report Size gives the size of the structure in bits. Report Count defines how many structures will be used. In the example descriptor above, the lines Report Size (8) and Report Count (6) define a 6 byte key array for a keyboard.

Collection items are used to show a relationship between two or more sets of data. For example, a minimal keyboard can be described as a collection of four data items (Modifier byte, Reserved byte, LED report, and Key Array). End Collection items simply close the collection.

It is important to note that all examples given here are merely for clarification. They are not necessarily definitive solutions.

A more detailed description of all items discussed here as well as other descriptor issues can be found in the “Device Class Definition for Human Interface Devices (HID)” revision 1.0 and in the “Universal Serial Bus Specification” revision 1.0, chapter 9. Both of these documents can be found on the USB world wide web site at http://www.usb.org/.

Functionality

Main Loop

The main loop of the firmware (*Figure 20*) waits for the keyboard Scan Task to be scheduled by the 1 ms ISR (*Figure 25*), and then executes it. Reports are sent to the host via End Point 1 anytime a key is pressed or released. The host side software will repeat the key press until the release of the key so that the device does not require a continuous report when no key status changes have occurred.

Scan Task

The Scan Task (*Figure 21*) determines if there were any key status changes from the previous scan (i.e. key presses or releases) and calls the KeyChanged routine (*Figure 22*) for each column of the scan matrix that had changes in any of the row positions. The KeyChanged routine calls the FoundKey routine (*Figure 24*) for each row that there was change in status. The FoundKey routine calculates an index based upon the row and column of the key, does a lookup in a KeyCodeTable for its usage (scan) code, and stores it in the End Point 1 FIFO. After all columns have been scanned, the Scan Task calls the SendKeys routine (*Figure 23*) if a report is required to be sent to the host.

Debounce

The mechanical switch properties of the key switches cause them to bounce after a key press. These bounces may be mistaken for actual key events. See *Figure 18*.

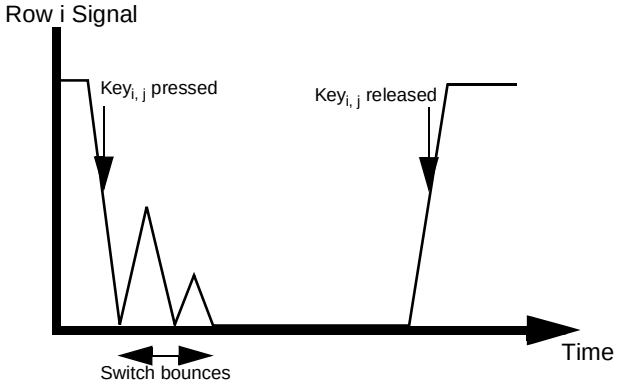

Figure 18. Key Switch Bounce

To solve this problem, all keys that are pressed have their code stored in a Debounce FIFO at the location pointed to by the Debounce Pointer (the reference firmware uses a 4-element FIFO). The 1 ms ISR updates and clears the current position in the Debounce FIFO every 12 ms. Every key that has a status change (i.e. pressed or released) is searched for in the Debounce FIFO. If the key is found, the status change is ignored. In other words, any status change for 12*4= 48 ms after a key is pressed is considered to be caused by switch bounces, and should be disregarded. Different keyboards will have varying key switch sensitivity, and the length of the Debounce FIFO may vary. The firmware uses the constant DEB_HI_ADDR to set the length of the FIFO.

Phantom Keys

A phantom fourth key press (D) will be detected in a scan matrix if two keys (A and B) in the same column are pressed, along with a third key (C) pressed in the same row as A or B (see *Figure 19*). If this situation is detected, all 6 key array positions in the End Point 1 FIFO (bytes 2–7) will be loaded with usage code 01h to report a *Rollover* error to the host. This tells the host that the firmware was unable to accurately determine which key presses had occurred.

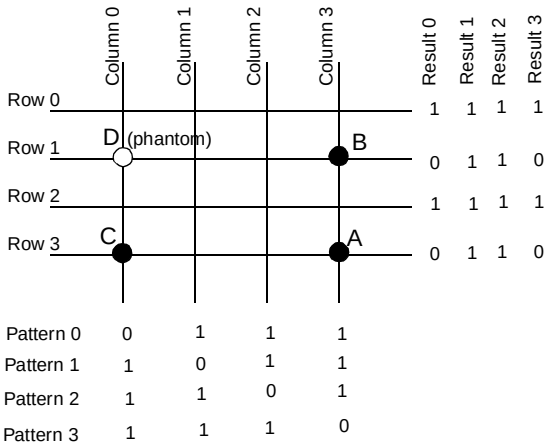

Figure 19. Phantom Key Situation

N-Key Rollover

In the case where multiple keys are pressed and held down, only the last key pressed should be repeated. To accomplish this, the last key press is stored in the *LastTx* buffer and is sent again in a separate report to the host whenever a multiple key report is sent. Once the same key is released, a report notifying it of the release is sent to the host (all 6 key array positions are filled with 0's).

Suspend/Resume/Remote Wakeup

According to the USB Specification 1.0, a device has to go into *suspend* mode after 3 ms of no bus activity. The 1 ms ISR determines when this condition occurs (by reading the Bus Activity bit, bit 3, of the USB Status and Control Register, 1Fh) and suspends the microcontroller. This is accomplished by setting bit 0 (the Run bit) and bit 3 (Suspend bit) of the Processor Status and Control Register (FFh). The microcontroller will resume operation if a GPIO interrupt occurs or bus activity resumes.

Prior to suspending the microcontroller, the 1 ms ISR pulls down all the column port lines and enables a falling edge interrupt on all Port 2 lines (Row port). If any key is pressed while in the suspended state, one of the row lines will be pulled low and trigger a GPIO interrupt. The GPIO ISR (*Figure 26*) will send a resume signal (force a K state where D+ is high and D- is low for 10ms) to wake up the host (remote wake up).

Key Code Table

The Key Code Table found at the end of the keyboard firmware must be filled in uniquely for each keyboard. Each keyboard has unique, proprietary key mappings to its scan matrix. Each location in the Key Code Table corresponds to a column/row intersection that must contain the usage code for the key in that position of the scan matrix. Unused locations should be filled with 00H.

The reference Key Code Table contains a unique location number in each column/row intersection, beginning with 01H for Column 0, Row 0 up to A0H for Column 19, Row 7. These location codes are useful for determining the mappings of keys to the scan matrix if appropriate documentation is not available. This is possible through the use of a USB bus analyzer tool to record DATA packets sent to the host with each key press. The contents of the recorded DATA packet point to exactly one location in the reference Key Code Table. The location number can then be replaced with the actual usage code of the key pressed. Usage codes for the USB keyboard can be found in Appendix A.3 of revision 1.0 of the Device Class Definition for Human Interface Devices (HID).

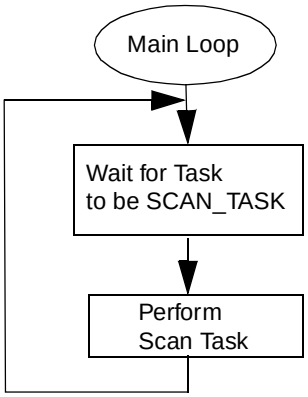

Figure 20. Main Loop

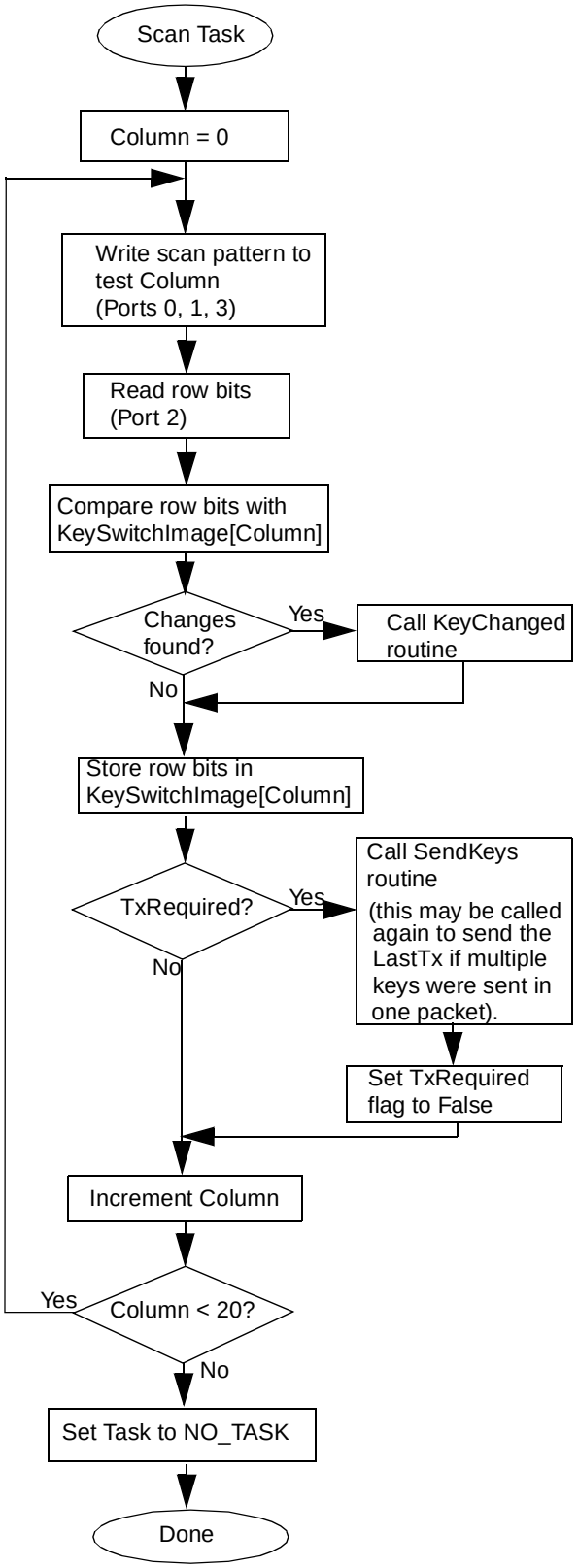

Figure 21. Scan Task

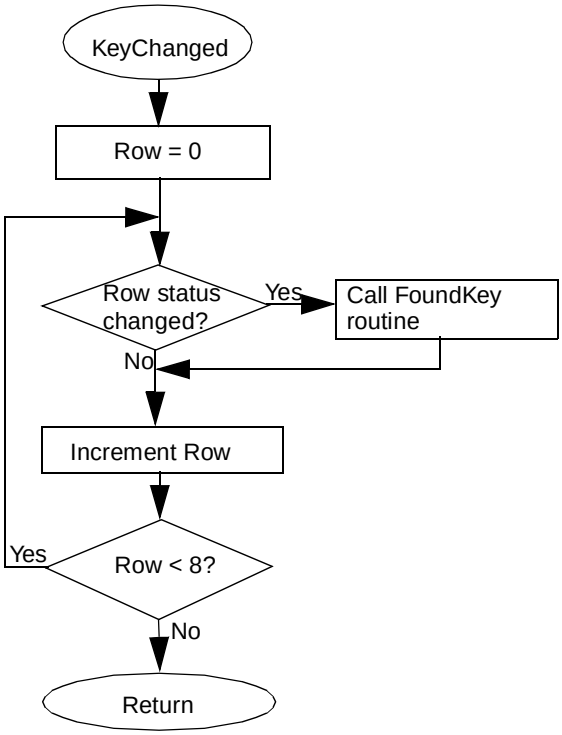

Figure 22. KeyChanged Routine

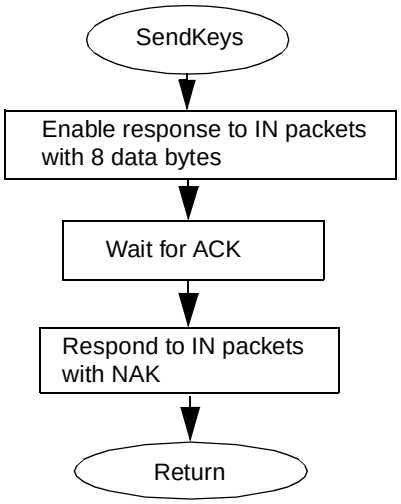

Figure 23. SendKeys Routine

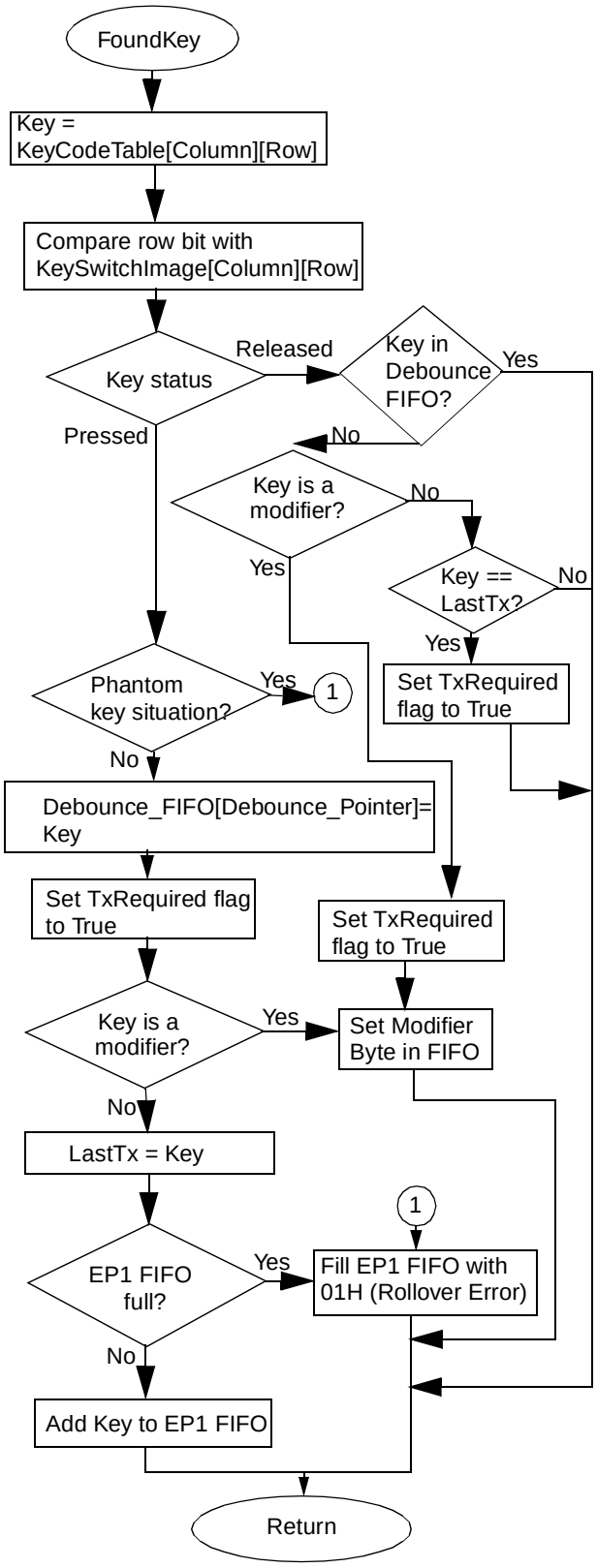

Figure 24. FoundKey Routine

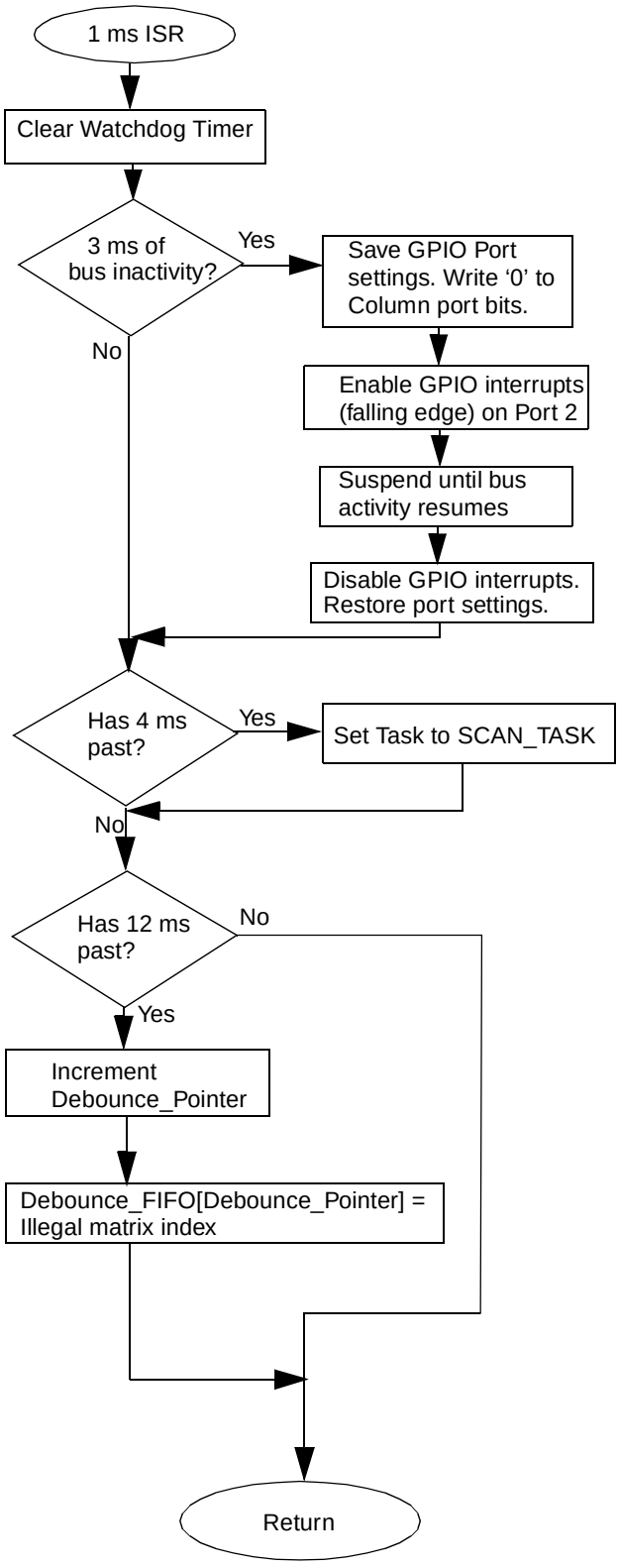

Figure 25. 1 ms ISR

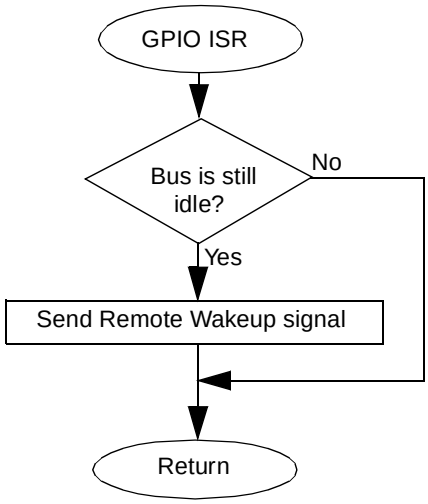

Figure 26. GPIO ISR

Conclusion

The two main enabling factors of the proliferation of the USB devices are cost and functionality. The CY7C63413 meets both requirements by integrating the USB SIE and multi-function I/Os with a USB optimized RISC core for a low price per part.