Designing a Low-Cost USB Interface for an Uninterruptable Power Supply with the Cypress Semiconductor CY7C63001 USB Controller

Introduction

The Universal Serial Bus (USB) is an industrial standard serial interface between a computer and peripherals such as a mouse, joystick, keyboard, UPS, etc. This application note describes how a cost-effective USB Uninterruptable Power Supply Interface (UPS) can be built using the Cypress Semiconductor single-chip CY7C63001 USB controller. The document starts with the basic operations of an uninterruptable power supply followed by an introduction to the CY7C63001 USB controller. A schematic of the CY7C63001 USB controller to the RS-232 of the UPS connection 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 USB UPS functions. Several sample code segments are included to assist in the explanation. Please contact your local Cypress sales office for a copy of the firmware.

This application note assumes that the reader is familiar with the CY7C63001 USB controller and the Universal Serial Bus. The CY7C63001 data sheet is available from the Cypress web site at www.cypress.com. USB documentation can be found at the USB Implementers Forum web site at www.usb.org.

USB UPS Basics

USB has been gaining popularity due to it's simple connection, plug and play feature, and hot insertion capability. This application note shows how an RS-232 UPS (low speed serial device) could be changed into a USB UPS using the CY7C63001 controller as shown in *Figure 1*.

In this design the UPS configuration and communication to the PC is all done through the USB interface. The RS-232 interface has a 2400 baud rate.

USB provides the plug-and-play feature that is not supported in RS-232 and PS/2 interfaces. The USB interface uses a four-pin connector with positive retention. A 28 AWG twisted pair is used for differential signaling and two 20 to 30 AWG wires are used to supply power and ground.

A simple UPS topology consists of a Battery system, a Power Converter system, a main AC input flow and an AC output flow. These are described in details in the "Report Descriptor" part of the "Firmware Implementation" section. For more details refer to the "Universal Serial Bus Device Class Definition and Usages Tables for Power Devices v. 0.9"

Introduction to CY7C63001

The CY7C63001 is a high performance 8-bit RISC microcontroller with an integrated USB Serial Interface Engine (SIE). The architecture implements 34 commands that are optimized for USB applications. The CY7C63001 has built-in clock oscillator and timers as well as programmable current drivers and pull-up resistors at each I/O line. High performance, low-cost human-interface type computer peripherals can be implemented with a minimum of external components and firmware effort.

Clock Circuit

The CY7C63001 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.

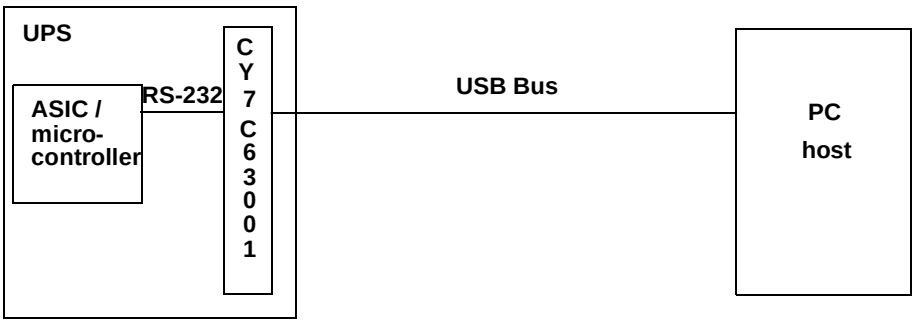

Figure 1. UPS to PC Connection

USB Serial Interface Engine (SIE)

The operation of the SIE is totally transparent to the user. In the receive mode, USB packet decode and data transfer to the endpoint FIFO are automatically done by the SIE. The SIE then generates an interrupt request 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 CY7C63001 has 12 general purpose I/O lines divided into 2 ports: Port 0 and Port 1. One such I/O circuit is shown in *Figure 2*. The output state can be programmed according to *Table 1* below**.** Writing a "0" to the Data Register will drive the output LOW and allow it to sink current.

Table 1. Programmable Output State

Port Data bit	Port Pull-up bit	Output State
0	X	Sink current "0"
1	0	Pull-up resistor "1"
1	1	High-Z

Instead of supporting a fixed output drive, the CY7C63001 allows the user to select an output current level for each I/O line. The sink current of each output is controlled by a dedicated 8-bit Isink Register. The lower 4 bits of this register contains a code selecting one of sixteen sink current levels. The upper 4 bits are reserved and must be written as zeros. The output sink current levels of the two I/O ports are different. For Port 0 outputs, the lowest drive strength (0000) is about 0.2 mA and the highest drive strength (1111) is about 1.0 mA. These levels are insufficient to drive LEDs.

Port 1 outputs are specially designed to drive high-current applications such as LEDs. Each Port 1 output is much stronger than their Port 0 counterparts at the same drive level setting. In other words, the lowest and highest drive for Port 1 lines are 3.2 mA and 16 mA respectively.

Each General Purpose I/O (GPIO) is capable of generating an interrupt to the RISC core. Interrupt polarity is selectable on a per bit basis using the Port Pull-Up register. Setting a Port Pull-Up register bit to "1" will select a rising edge trigger for the corresponding GPIO line. Conversely, setting a Port Pull-Up register bit to "0" will select a falling edge trigger. The interrupt triggered by a GPIO line is individually enabled by a dedicated bit in the Port Interrupt Enable registers. All GPIO interrupts are further masked by the Global GPIO Interrupt Enable bit in the Global Interrupt Enable register

The Port Pull-Up registers are located at I/O address 0x08 and 0x09 for Port 0 and Port 1 respectively. The Data registers are located at I/O address 0x00 and 0x01 for Port 0 and Port 1 respectively. The Port 0 and Port 1 Interrupt Enable registers are at addresses 0x04 and 0x05 respectively.

Hardware Implementation

The UPS USB interface is implemented as shown in *Figure 6*. A 7.5-KΩ resistor is used to pull up the D– line to 5V. This signals to the host that this is a low speed device upon plug-in. The interface to the RS-232 is done through two GPIO pins where bit banging is used.

RS-232C Electrical Characteristics

This design implements the following electrical characteristics:

- ±10 VDC Signaling Rails
- Three-wire Interface
  - Transmit Data
  - Receive Data
  - Signal Ground
- No Hardware Handshake

The use of an RS-232 level translator, that requires a single +5 VDC supply, provides a simple and effective hardware interface to the serial bus. This device uses voltage doubling and inversion techniques to provide the ±10 VDC signaling rails required by this design. The schematic for this device connection is shown in *Figure 7*.

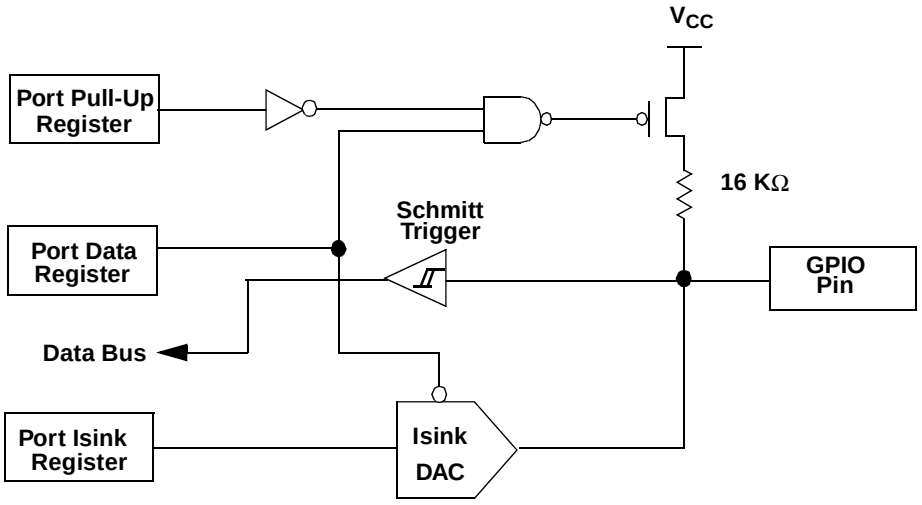

Figure 2. One General Purpose I/O Line

A simple three wire interface minimizes the hardware requirements and reduces the complexity of the firmware design.

RS-232C Serial Data Transfer Protocol

This design supports the following features:

- 2400 Baud
- 10-Bit Frame

  1 Start, 8 Data, 1 Stop
- No Parity
- Half Duplex Mode

By limiting the supported protocol to a single set of features, implementation of the design is simplified and data stability is ensured.

At the 2400-baud data rate, the design produces a 10-bit frame in 4.17 milliseconds thus transferring an ASCII character approximately every 5 milliseconds.

The data is “framed” by a beginning active LOW start bit and an ending active HIGH stop bit.

The use of 8 data bits provides access to the full range of the 256 character Extended ASCII set.

The data is transmitted onto the serial data bus starting with the least significant bit and progressing to the most significant bit.

Currently the design does not support parity checking.

The half duplex mode of communications supports exclusive transmit and receive operations only. Therefore data flow direction on the bus must be predetermined.

An ASCII Carriage_Return/Line_Feed character pair is appended to the end of each character string to signal the end of transmission.

Figure 3 illustrates the timing relationships of the elements that compose a serial data frame.

Serial Interface Transmit Protocol

The transmit routine uses the 128-µs interrupt, generated by the built-in timer, to generate the outgoing serial data stream.

During the transmit operation a character string is placed in the transmit buffer and the ASCII Carriage_Return/Line_Feed character pair is appended to end.

The first character is copied into the transmit register and the transmit routine is called.

A low start bit is generated on the first occurrence of the 128-µs interrupt and the data is then shifted onto the bus, LSB first, based on a count of the number of interrupts that have occurred. After the MSB of the data has been transmitted, an active HIGH stop bit is driven onto the bus to complete the frame. The bus is continually driven HIGH until the start bit of the next frame is transmitted.

Each character is then copied in turn to the transmit register and the transmit routine is called repeatedly until the transmit buffer is empty.

Upon completion of the transmit operation, the main routine enables the GPIO interrupts and waits for a response in the receive data line.

Figure 4 illustrates the use of the 128-µs interrupt to establish a serial transmit data frame. Timing is based on the hexadecimal count of the number of interrupts from the start of the transmit frame.

Serial Interface Receive Protocol

The receive routine uses the GPIO interrupt and the 128-µs interrupt, generated by the built-in timer, to process the incoming serial data stream.

When the receive routine is called, GPIO interrupts are enabled, allowing a received start bit to signal the beginning of the first data frame.

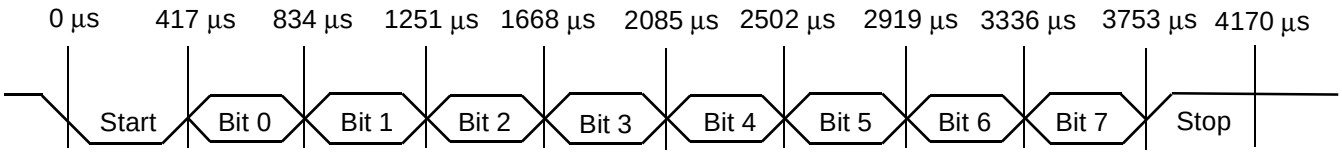

Figure 3. RS-232C Serial Data Frame

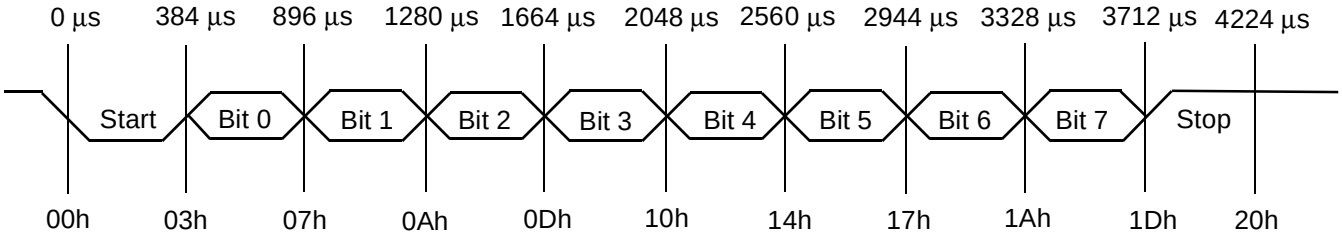

Figure 4. Transmit Serial Data Frame

The start bit is synchronized to the subsequent 128-µs interrupt and checked for validity. The following data bits are sampled, based on the count of 128-µs interrupts, at the near midpoint of the frame and shifted into the receive register. The stop bit is sampled and checked for validity.

If a valid frame (start bit = '0', stop bit = '1') is detected, the receive register contents are copied into the receive buffer. If an invalid frame is detected a flag is set indicating a framing error and the receive operation continues.

The received data is compared to the value of the ASCII Line_Feed character (0Ah). If the Line_Feed character is detected, the receive operation ends and the firmware returns to the calling routine. If the Line_Feed character is not detected the receive operation continues to accept characters and place them in the receive buffer.

The ASCII Carriage_Return character (0Dh) is retained, as a string terminator, in the receive buffer for further data processing.

If an invalid frame was detected, as indicated by the framing flag, the receive buffer is reinitialized and the transmit/receive cycle is repeated.

Figure 5 illustrates the use of the 128-µs interrupt to establish a serial receive data frame.

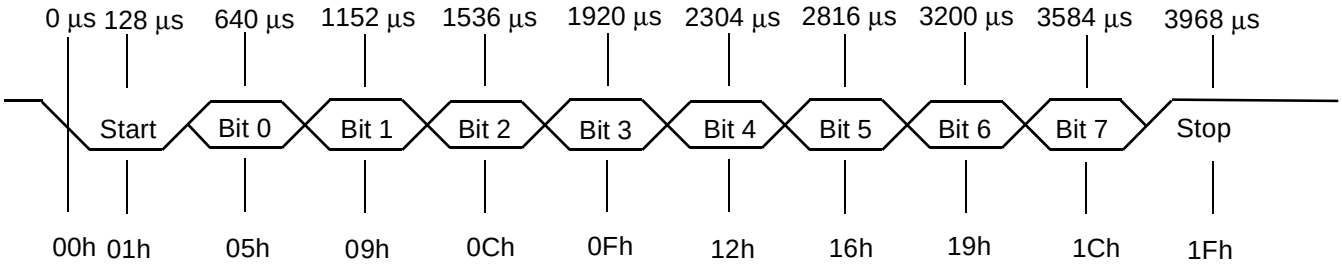

Figure 5. Receive Serial Data Frame

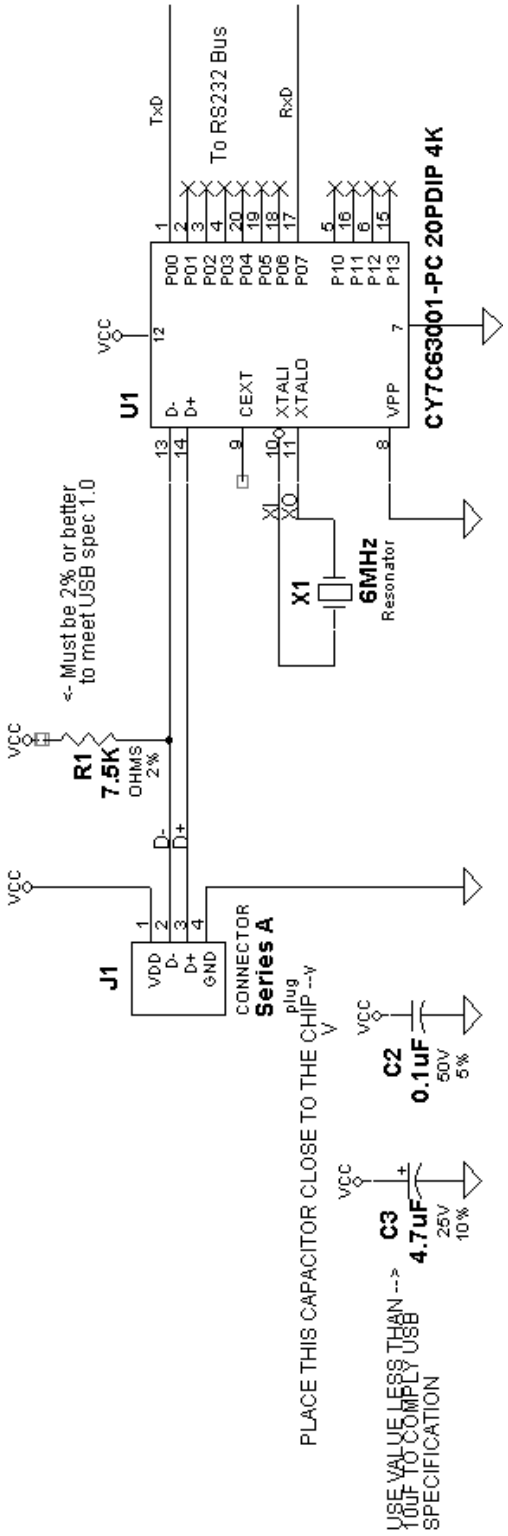

Figure 6. Hardware Implementation

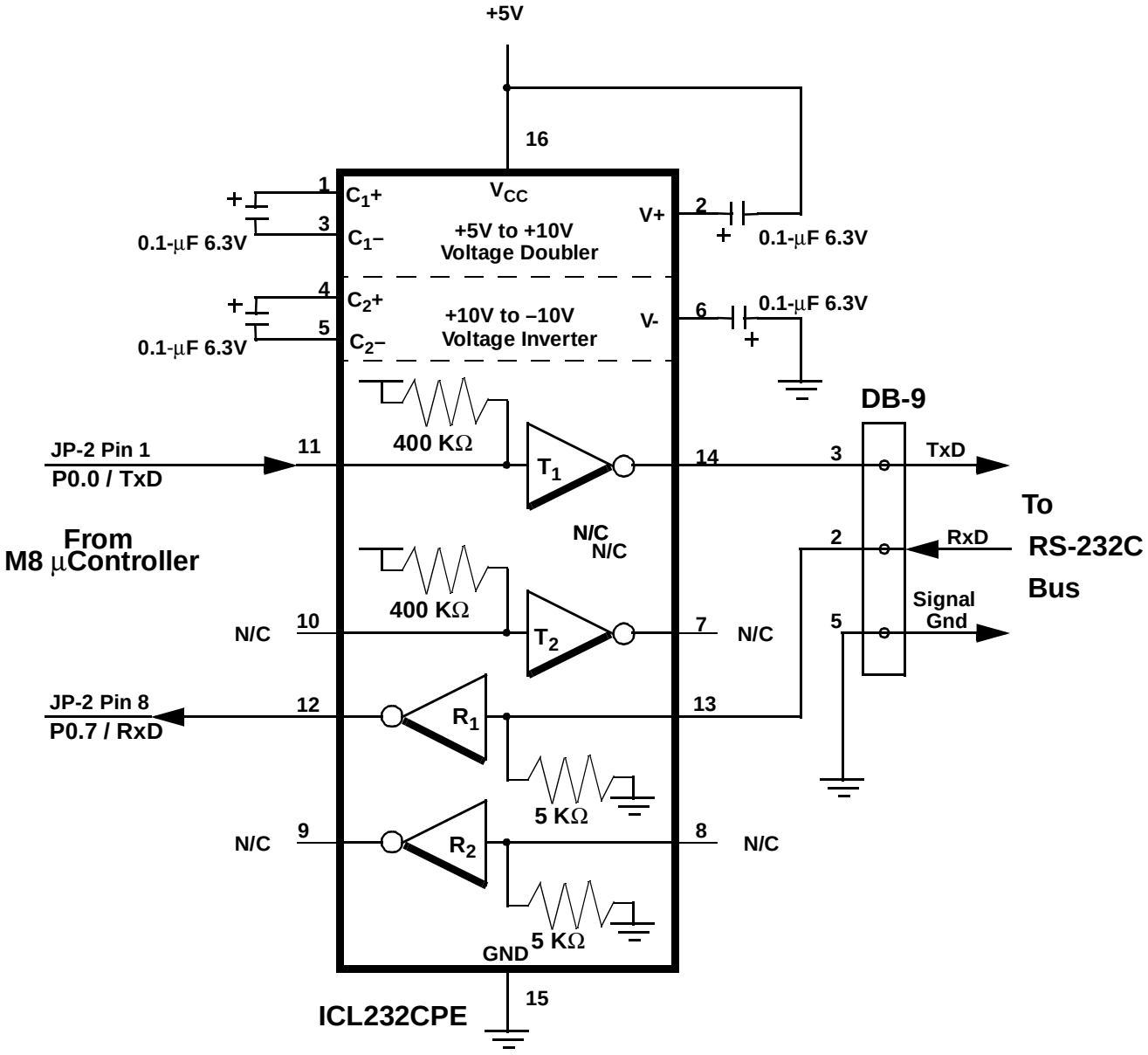

Figure 7. Electrical Schematic Diagram—Serial Interface Modification

Firmware Implementation

USB Interface

All USB Human Interface Device (HID) class applications follow the same USB start-up procedure. The procedure is as follows (see *Figure 8*):

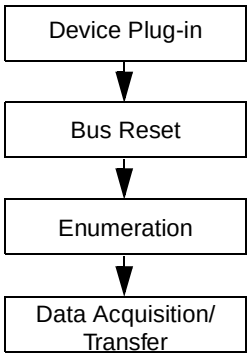

Figure 8. USB Start-Up Procedure

Device Plug-in

When a USB device is first connected to the bus, it is powered but remains non-functional waiting for a bus reset. The pull-up resistor on D– notifies the hub that a low-speed (1.5 Mbps) device has just been connected.

Bus Reset

The host recognizes the presence of a new USB device and resets it (see *Figure 9*).

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, while the host continues to ask for information about the device description, configuration description and HID report description. Using the information returned from the device, the host now knows the number of data endpoints supported by the device. At this point, the process of enumeration is completed. See *Figures 10, 11,* and *12*

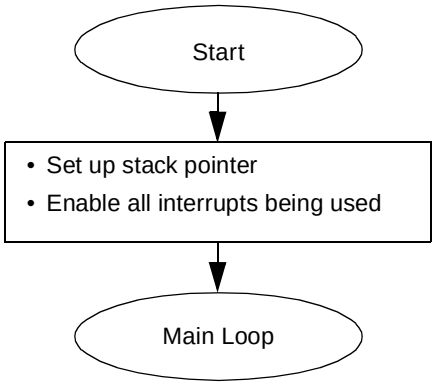

Figure 9. Reset Interrupt Service Routine

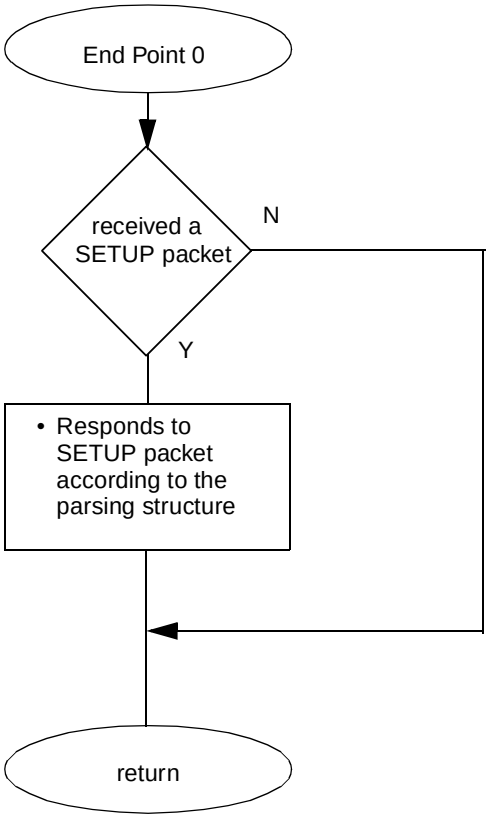

Figure 10. Endpoint 0 ISR

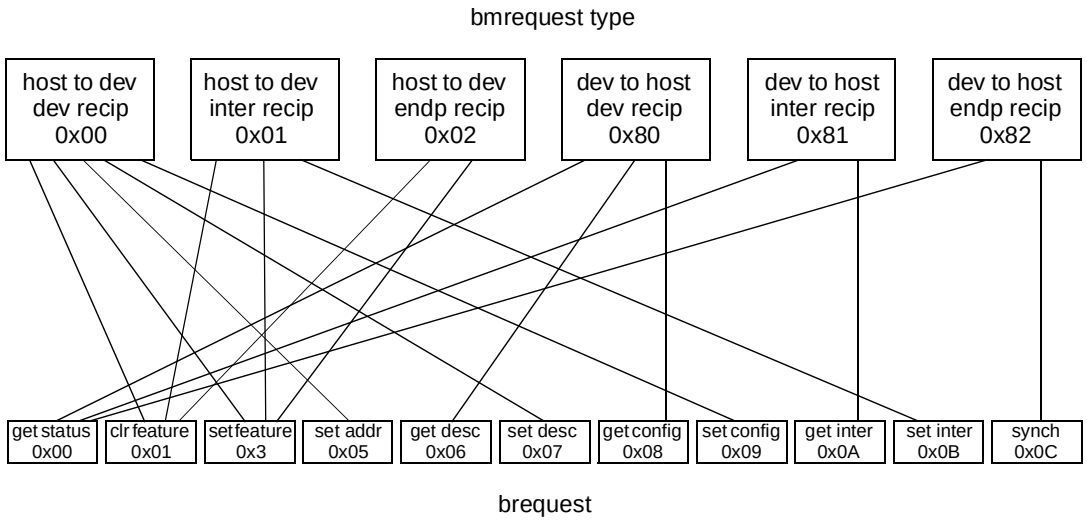

Figure 11. USB Standard Request Parsing Structure

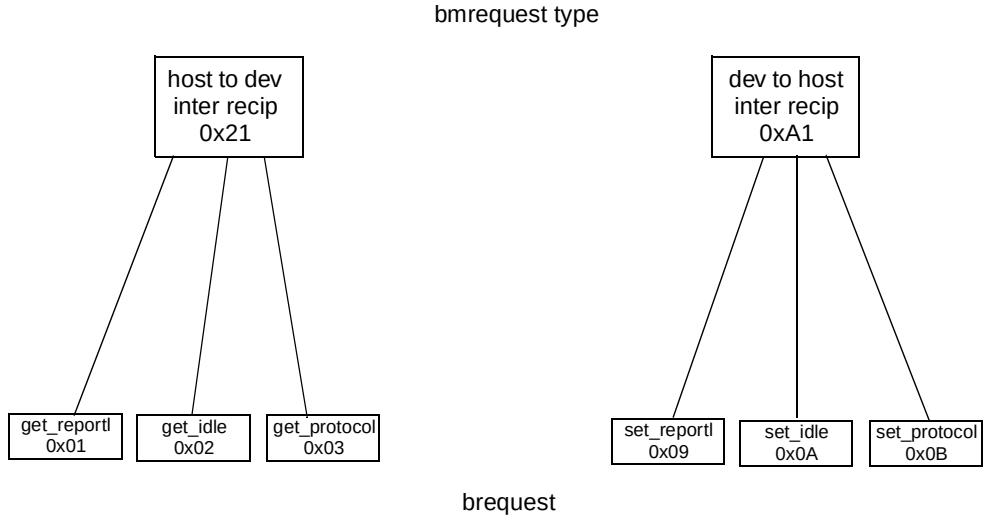

Figure 12. USB HID Class Request Parsing Structure

Data Acquisition/Transfer

Data Transfer is done through several reports (five in our case). When the host asks for one of these reports the device translates the request into a set of UPS commands and sends them across the RS-232 bus to the UPS. After the UPS replies to all the commands the controller converts the response to the required format. When the host issues IN packets to retrieve data from the device, the device returns the converted and formatted data to the host. The host asks for the reports through the *get report* request. Each report has a unique ID. The subroutine that implements the *get report* request is shown in *Figure 15*.

The *set report* request ID 3 is used to test the UPS. When this request is received from the host the microcontroller sends a test command to the UPS over the RS-232 interface. This command starts the testing process on the UPS. A *get report* request ID 3 should then be issued by the host to be able to read the results of the tests performed. The *set report* subroutine is implemented as shown in *Figure 16.*

Therefore, all data transfer between the host and the UPS through the CY7C63001 microcontroller is done through endpoint 0 using the different reports. Since this is a low-speed device, the maximum transfer rate is 8 bytes per ms.

The file Serial.asm contains the subroutines necessary to communicate with the peripheral. The transmit, receive, delay and various support subroutines are located here.

Further clarification of code functionality is included as comments throughout the assembly source code.

Firmware Flow for Transmit Routine

Figure 13 illustrates the flow of the assembly code for the transmit routine of the Serial Interface design.

The routine is initially called to transmit the first character in the receive buffer. It will continue to loop, sending further characters, until it has reached the end of the buffer.

Note that the buffer terminates with an ASCII carriage_return/line_feed pair. This indicates the end of data transmission to the receiving device.

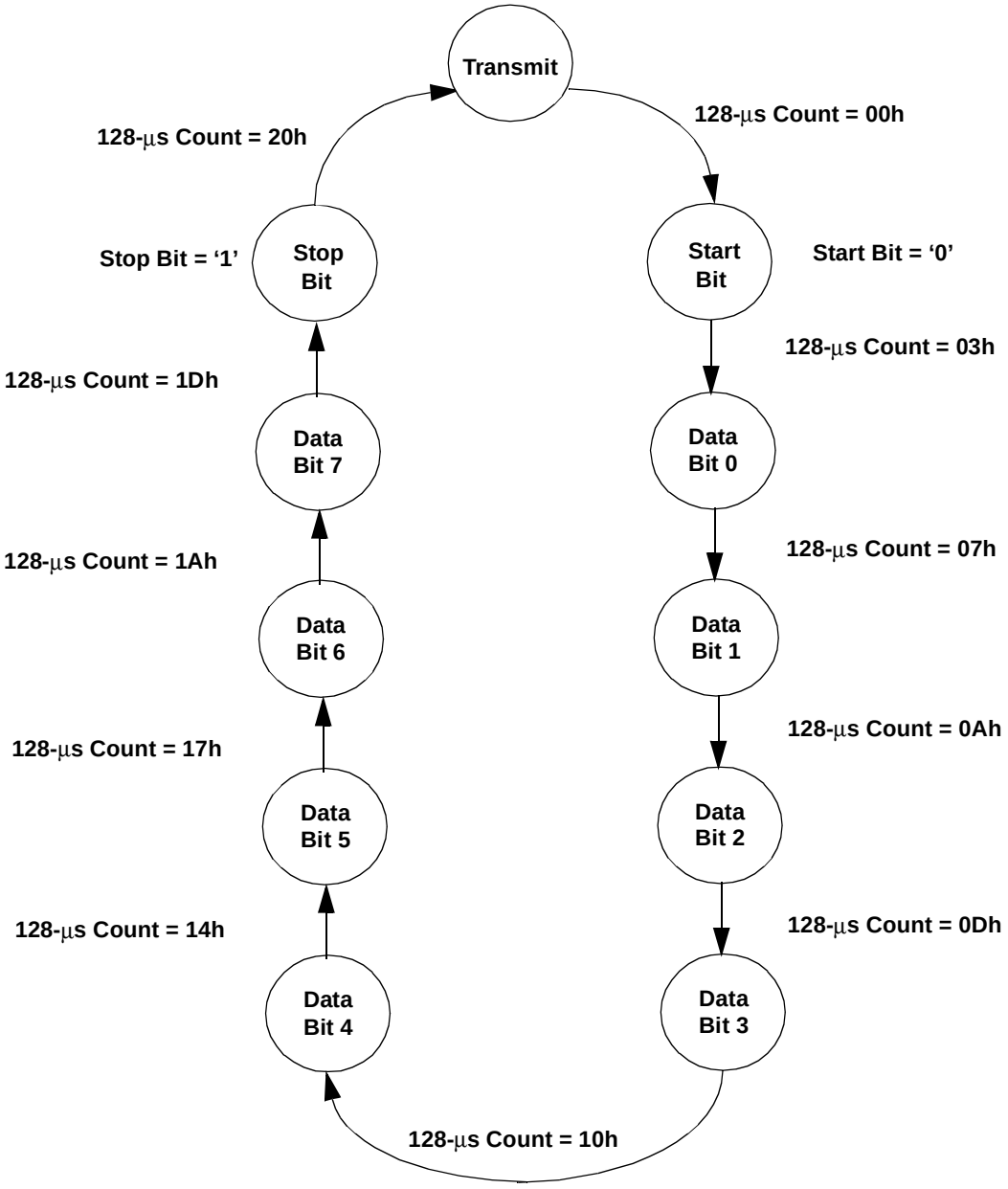

Figure 13. Firmware Flow Diagram - Transmit

Firmware Flow for Receive Routine

Figure 14 illustrates the flow of the assembly code for the transmit routine of the Serial Interface design.

The routine is initially entered after transmitting a command on the RS-232 bus. The responding device begins transmitting the requested data as an ASCII character string. The receive routine continues to loop, accepting characters, until it detects an ASCII carriage_return/line_feed pair. All received characters are saved in a buffer except for the line-feed character which is stripped. If a framing error occurs, the receive buffer will be flushed and the previous command will be reissued to the responding device.

The resulting receive buffer contains the ASCII character string terminated with the carriage_return character. This limits the return character string to 15 characters in length.

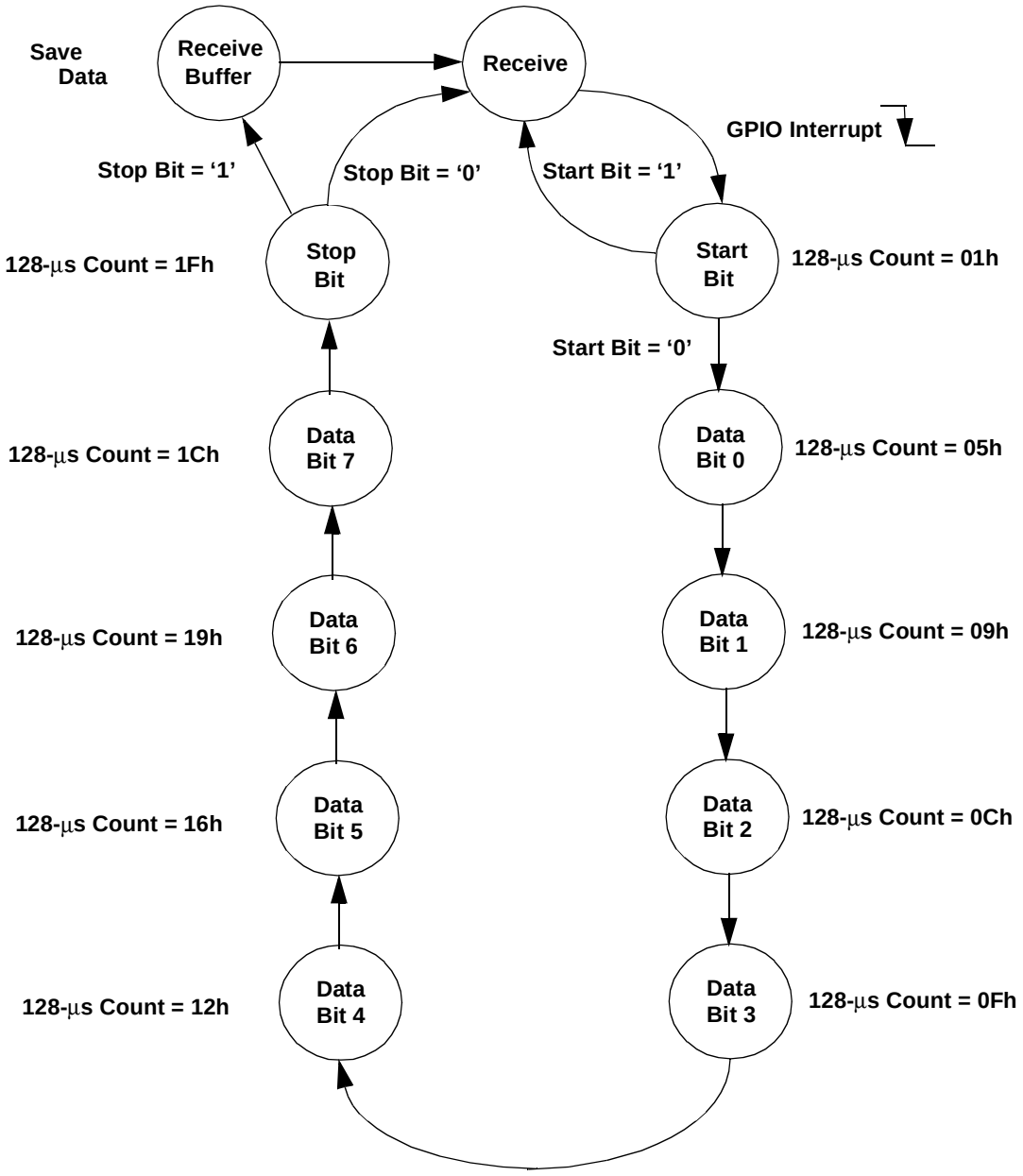

Figure 14. Firmware Flow Diagram - Receive

Care should be taken to avoid accepting more than 15 characters as the memory area succeeding the receive buffer is used by the microcontroller's data stack. Should the peripheral send more than 15 characters in a string, or if a large data

stack is required, the data memory space should be adjusted to increase the area between the receive buffer and the data stack pointer.

Note that the carriage-return character is retained as a delimiter for further processing of the receive buffer data.

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 UPS)
Device Release Number (1.03)
String Describing Vendor (1)
String Describing Product (2)
String for Serial Number (3)
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 (Self 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)
```

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)
```

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)
```

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.

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.

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. There is also another kind of Usage tag 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.

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.

Collection items are used to show a relationship between two or more sets of data. End Collection items simply close the collection.

In this UPS implementation the report descriptor contains the following collections:

- Main AC Flow Physical Collection
- Output AC Flow Physical Collection
- Battery System Physical Collection
- Power Converter Physical Collection
  - — AC Input Physical Collection
  - — AC Output Physical Collection

Each one of these collections is sent to the host through a different report and therefore each has a different report id.

In this UPS implementation these collections contain the following objects as well as the report ids:

- The Main AC Flow Physical Collection contains:
  - — Flow ID
  - — Configuration Voltage
  - — Configuration Frequency
  - — Low Voltage Transfer (the minimum line voltage allowed before the UPS system transfers to battery backup)
  - — High Voltage Transfer (the maximum line voltage allowed before the UPS system transfers to battery backup)
  - — Manufacturer Name Index
  - — Product Index
  - — Serial Number Index
- The Output AC Flow Physical Collection contains:
  - — Flow ID
  - — Configuration Voltage
  - — Configuration Frequency
  - — Configuration Apparent Power
  - — Configuration Active Power (RMS)
  - — Delay Before Startup
  - — Delay Before Shutdown
- The Battery System Physical Collection contains:
  - — Battery System ID
  - — Present Status (Used, Good)
  - — Voltage
  - — Temperature
  - — Test
- The Power Converter Physical Collection contains:
  - — Power Converter ID
  - — AC Input Physical Collection
  - — AC Output Physical Collection
- The AC Input Physical Collection contains:
  - — Input ID
  - — Flow ID
  - — Present Status (Good)
  - — Voltage
  - — Frequency
- The AC Output Physical Collection contains:
  - — Flow ID

— Voltage

— Frequency

— Percent Load

— Present Status (Overload, Boost, Buck)

An example of part of a report descriptor for a UPS can be found below.

```
Example of part of the report descrip-
tor

Usage Page (Power Device)
Usage (UPS)
Collection (Application)
     Usage Page (Power Device)
     Usage (Flow)
     Collection (Physical)
          ReportID (1)
          Usage (ConfigVoltage)
          Report Size (16)
          Report Count (1)
          Unit (Volt)
          UnitExponent (7)
          Logical Minimum (0)
          Logical Maximum (250)
          Feature  (Data,Variable,Ab-
solute)
     End Collection
End 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, “Universal Serial Bus Device Class Definition and Usage Table for Power Devices” revision 0.9a 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/.

String Descriptors

String descriptors are used to specify any strings that need to be sent to the host. They could be used for manufacturer name, product and serial number, etc. They are optional and are UNICODE encoded. An example of a string descriptor is given below.

```
Example of string descriptor

Descriptor Length (in bytes)
Descriptor Type (string)
String
```

In this application string descriptors are implemented as dynamic except the language descriptor which is static and set to “U.S. English”. Dynamic descriptors are variable depending on the UPS connected. Data is sent from the RAM not from the ROM. This allows us not to change the code when connecting to a different UPS. The strings supported are the Manufacturer name, product and serial number.

When the host sends a packet asking for one of the string descriptors the controller will poll the UPS on the RS-232 bus and get the string then send it to the host in the right format.

Note that unlike report descriptors that only describe the format of the data sent through reports, the string descriptors actually include the string itself (there is no get string request). The *get string descriptor* subroutine implementation is shown in *Figure 17.*

Conclusion

USB has been gaining popularity due to its simple connection, plug and play feature, and hot insertion capability. The two main enabling factors of the proliferation of the USB devices are cost and functionality. The CY7C63001 meets both requirements by integrating the USB SIE and multi-function I/Os with a USB optimized RISC core. This application note allows designers to easily convert an RS-232 UPS interface to a USB UPS interface

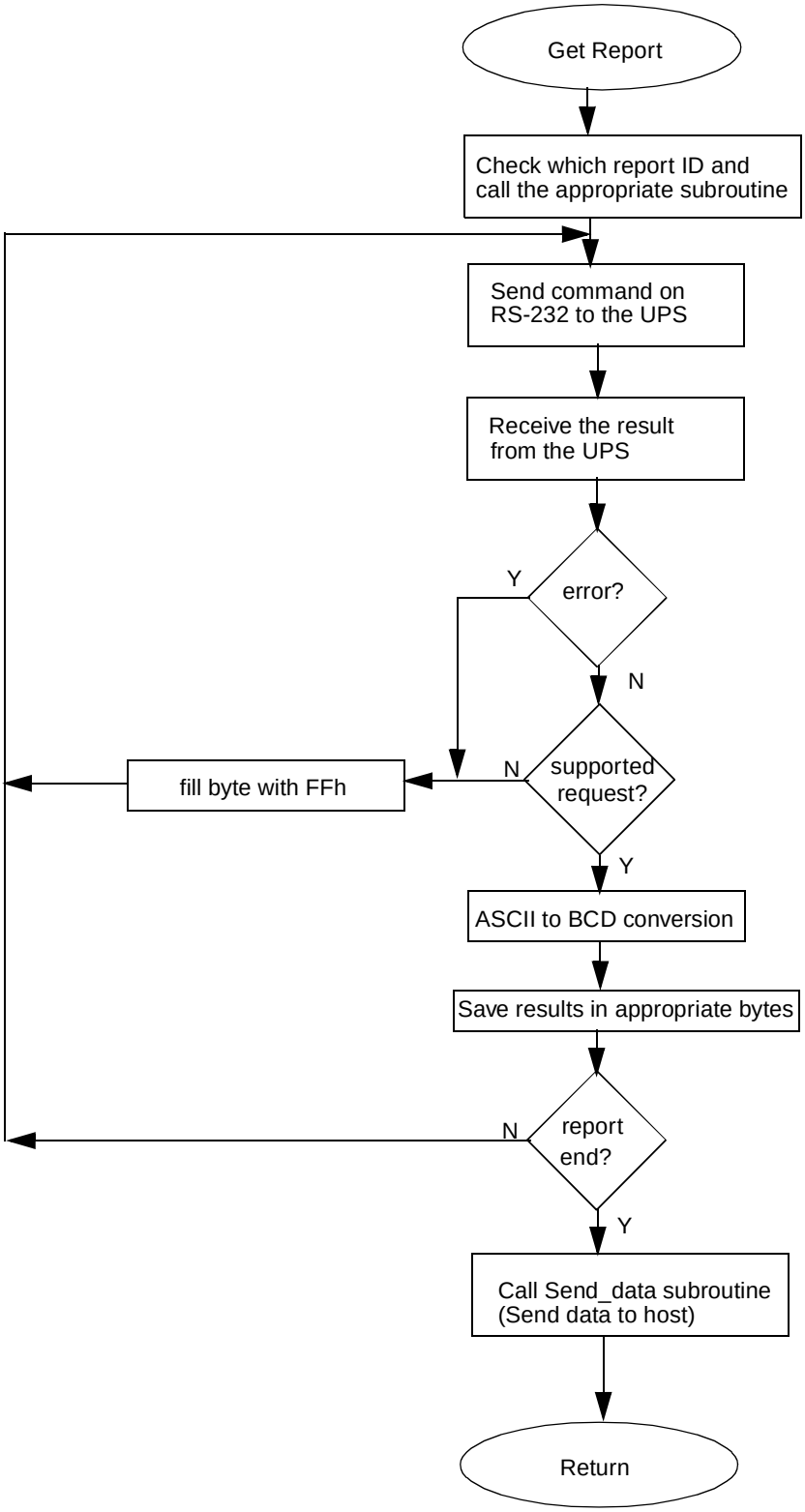

Figure 15. Get Report Subroutine

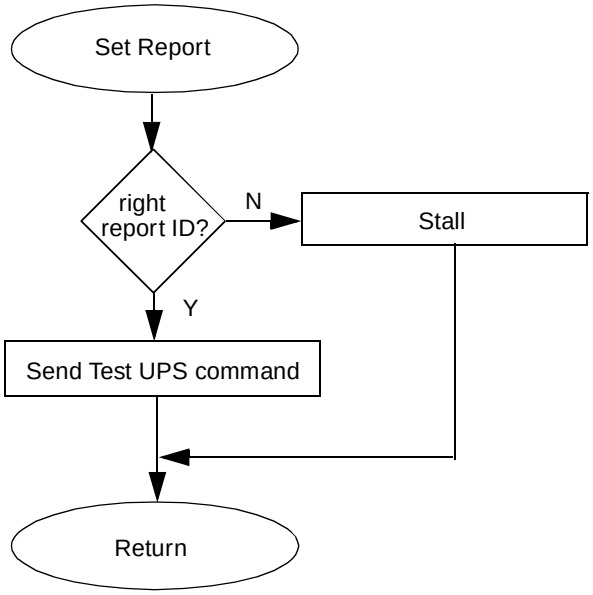

Figure 16. Set Report Subroutine

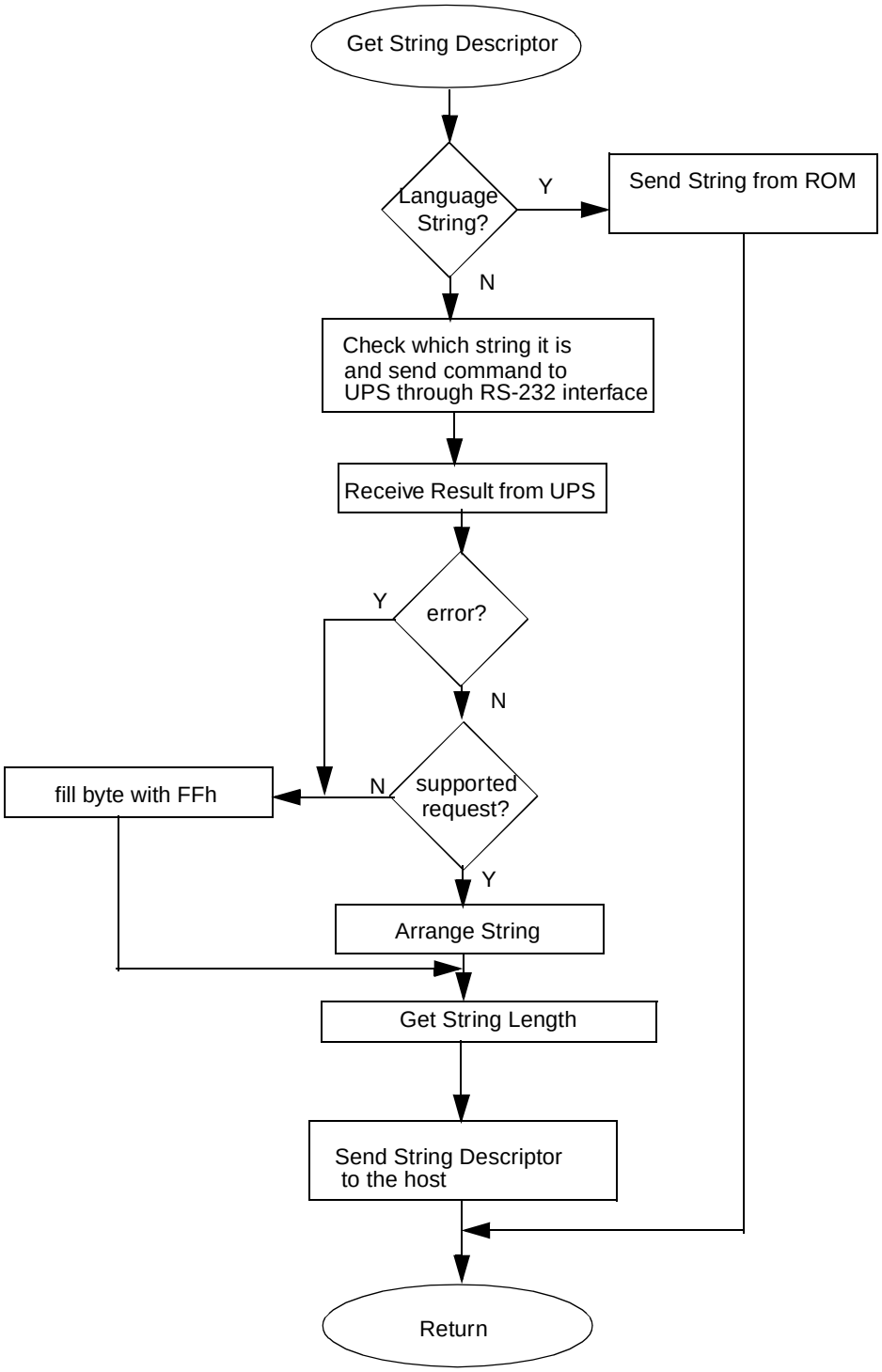

Figure 17. Get String Descriptor Subroutine