CY4636 Reference Design Kit Guide

Version 1.1 January 22, 2007

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com

Copyrights

Disclaimer

Contents

1. Introduction

1.1 Scope

The audience for this document is intended to be firmware and hardware developers that want to understand and make modifications to the WirelessUSB™ LP KBM Reference Design Kit (RDK).

1.2 Overview

This document provides a description of the hardware along with architecture and configuration options for the WirelessUSB LP KBM RDK.

1.3 Definitions

Following are some definitions of acronyms and words found in this document. There may be other meanings to these definitions outside of this document.

Bridge – The bridge is the receiving radio and USB hardware that connects to the PC and enumerates as a Human Interface Device.

Device – The reference to device in this document means the keyboard or mouse device that is sending radio packets to the bridge.

DVK – A development kit produced by Cypress Semiconductor for showcasing Cypress products with a working development environment.

HID – Stands for Human Interface Device and is a product that allows an individual to interface with a computer. A keyboard and mouse are HID devices.

RDK – A reference design kit produced by Cypress Semiconductor and used by 3rd parties to produce off-the-shelf products. Everything required to take a product to production is included in the kit. This document is part of the CY4636 Mouse/RDK keyboard.

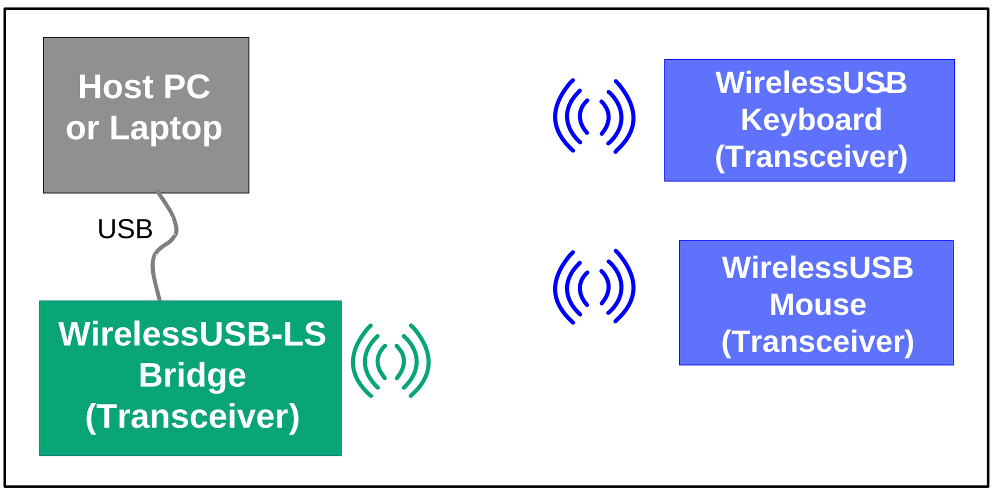
USB – The acronym for Universal Serial Bus, a well-known serial standard used in the computing world.

WirelessUSB™ – a trademark name for Cypress 2.4 GHz radio products.

2. WirelessUSB 2-Way HID Protocol Overview

The WirelessUSB 2-way Human Interface Device (HID) protocol is designed for reliable 2-way communication between a wireless bridge and target HID device in 1:1 (one HID and one bridge) or 2:1 (two HIDs and one bridge) systems. The WirelessUSB 2-way HID protocol allows HID applications to establish a connection to the bridge and receive ACK and DATA packets from the bridge. The host PC is not aware of the wireless connection, since the interface to the host acts like a normal wired USB HID connection. Therefore, there is no special software required on the host PC in order to support WirelessUSB.

Figure 2-1. WirelessUSB 2-Way System



2.1 Radio Channel Management

WirelessUSB uses the unlicensed 2.4 GHz Industrial, Scientific, and Medical (ISM) band for wireless connectivity. WirelessUSB uses 78 of the available channels and splits the 78 channels into 6 channel subsets consisting of 13 channels each. The channel subsets are used by each network to minimize the probability of interference from other WirelessUSB systems (see the *Channel Selection Algorithm* section for more details). A designated channel subset is used during Bind Mode (along with an associated pseudo-noise code) in order to enable all WirelessUSB devices to effectively communicate during this procedure.

2.2 Pseudo-Noise Codes

Pseudo-noise codes (PN codes) are the codes used to achieve the special matched filter characteristics of DSSS communication. Certain codes referred to as 'multiplicative codes' are used for WirelessUSB 2-way communication. These codes have minimal cross-correlation properties, meaning they are less susceptible to interference caused by overlapping transmissions on the same channel. The length of the PN code results in different communication characteristics. Higher data rates are achieved with 32-chips/bit PN codes, while 64-chips/bit PN codes allow a longer range. The number of frequency/code pairs is large enough to comfortably accommodate hundreds of WirelessUSB devices in the same space. Each bridge/HID pair must use the same PN code and channel in order to communicate.

2.3 Chip Error Correction

In the presence of interference (or near the limits of range), the transmitted PN code are often received with some PN-code chips corrupted. DSSS receivers use a data correlator to decode the incoming data stream. WirelessUSB LP supports a separate SOP and Data threshold. The RDK uses an SOP threshold of ‘4’. The data threshold is set to the default value of ‘4’.

2.4 Automatic Acknowledgment

The WirelessUSB LP radio contains an automatic acknowledgment (AutoACK) feature that allows it to automatically send an ACK to any valid packet that is received. The WirelessUSB LP radio also uses the concept of transactions to allow the radio in the HID to automatically power down after transmitting a packet and receiving an AutoACK, instead of waiting for the firmware to power the radio down. This conserves power and reduces the firmware complexity of WirelessUSB applications.

2.5 Network ID

The Network ID contains the parameters for the Channel Selection Algorithm as well as the PN code to be used. HIDs retrieve the Network ID information from the bridge during the Bind Procedure. A special Network ID is reserved for Bind Mode, known as the Bind ID. The Bind ID gives a common channel subset so that any two devices can communicate with each other during Bind Mode. The Network ID is composed of the following fields:

PIN – This is a random number, between 2-5, that defines the Channel subset and is used in the Channel Selection Algorithm.

Base Channel – This is the first channel to be used in the Channel Selection Algorithm, that determines which channels are contained in the channel subset.

PN Code – This is used as an index to select one of 10 used SOP PN codes, as noted in the radio driver.

CRC Seed – This 8-bit value is used for the CRC calculation, that further diversifies transmissions from different networks. All packets sent between non-bound devices use the default CRC seed of 0x0000. All packets sent between bound devices use a CRC seed that is common to all devices bound to a particular bridge or network but unique from network to network.

2.6 Manufacturing ID

Each WirelessUSB radio contains a 4-byte Manufacturing ID (MID), that has been laser fused into the device during manufacturing. The bridge uses its MID to help randomize channel subsets, PN codes and Network CRC Seeds. The bridge sends its MID to the HIDs when binding. The HID then stores the bridge’s MID in non-volatile memory after binding. The HID sends the bridge’s MID as part of the Connect Request packet, allowing the bridge to verify the identity of the HID when establishing a connection.

Both the bridge and the HID use the bridge's MID to generate the device Network ID components. The following equations ensure that each network will have a unique set of Network ID components:

PN Code = (mid_1 << 2) + mid_2 + mid_3

Base Channel = (mid_2 >> 2) **-** (mid_1 << 5) + mid_3

PIN = (((mid_1 -mid_2) & PIN_MASK) + MIN_PIN)

CRC Seed = ((mid_2 >> 6)) + mid_1 + mid_3

2.7 Channel Selection Algorithm

The Channel Selection Algorithm produces a subset containing 13 of the possible 78 channels. The Channel Selection Algorithm is based on the Network ID, with each channel in the subset six megahertz from the nearest neighboring channels in the subset. This algorithm reduces the possibility of multiple bridges selecting the same channels in the same order at the same time.

2.8 Protocol Modes

The ping, idle, reconnect, bind, and data modes are discussed in this section.

2.8.1 Ping Mode (Bridge Only)

Ping Mode is used by the bridge to find an available channel; channels are unavailable if they are being used by another network with the same PN code, or if there is excessive noise on the channel. The bridge first listens for activity on the selected channel. If the channel is inactive the bridge alternately transmits Pings and listens for Ping Responses for a defined* period of time. During Ping Mode the bridge also checks the Receive Signal Strength Indicator (RSSI) of the radio in order to determine if a non-WirelessUSB device is using this channel (or a WirelessUSB device on the same channel using a different PN code). If a Ping Response is received, indicating that another bridge is using this channel the bridge selects the next channel using the Channel Selection Algorithm and repeat this procedure. The bridge also selects another channel using the Channel Selection Algorithm if RSSI is high; this indicates that there is other RF sources on the channel. If a Ping Response is not received and RSSI is low, the bridge assumes the channel is available and moves to Data Mode. Bridges send Ping Responses in response to all received Pings if the bridge is in Data Mode. HIDs never respond to Pings.

[*The timeout value is configurable using the PING_NUM_RSSI define.]

2.8.2 Idle Mode (HID only)

This is the state of an HID after a power-on reset before it has had any communication with the WirelessUSB bridge. If the bridge's MID is stored in nonvolatile memory the HID retrieves the bridge's MID, calculate the Network ID and move to Reconnect Mode. If the bridge's MID is not stored in nonvolatile memory the HID waits in Idle Mode until a user-initiated event causes the HID to enter Bind Mode. After a defined period of time in Idle Mode the HID enters Sleep Mode in order to conserve power. When the HID wakes up due to a user action, it re-enters Idle Mode.

2.8.3 Reconnect Mode (HID only)

Reconnect Mode is used by the HID to discover the current channel used by the bridge and to establish a connection with the bridge. Upon entering Reconnect Mode the HID uses the Network ID to select a channel using the Channel Selection Algorithm. The HID transmits Connect Requests containing the Manufacturing ID of the desired bridge and listens for an AutoACK. If an AutoACK is received the HID disables the AutoACK and continues to listen for a Connect Response. If a bridge in Data Mode receives a Connect Request containing its Manufacturing ID, it sends a positive Connect Response to the HID. If a HID receives a positive Connect Response it moves to Data Mode. If a HID does not receive a positive Connect Response, it selects the next channel using the Channel Selection Algorithm and repeats the procedure. If the HID does not receive a positive Connect Response on any of the channels in the subset, it enters Sleep Mode in order to conserve power. When the HID wakes up due to a user action it reenters Reconnect Mode.

2.8.4 Bind Mode

2.8.4.1 HID

Bind Mode allows the HID to retrieve the bridge's Manufacturing ID which is used to calculate the Network ID. Upon entering Bind Mode the HID sets the current channel and PN code to the channel and PN code specified in the Bind ID. The HID then transmits Bind Requests and listens for an AutoACK. If an AutoACK is received, the HID (keeping the AutoACK enabled) continues to listen for a Bind Response (containing the bridge's MID) from the bridge. If a Bind Response is not received the HID moves to the next channel. If a Bind Response is received the HID stores the bridge's MID, calculates the Network ID and moves to Reconnect Mode. The algorithms used to calculate these fields are implementation specific and should be the same on the bridge and the HID (both devices use the bridge's Manufacturing ID to calculate these fields). If a defined* period of time has elapsed while in Bind Mode without receiving a Bind Response, the HID exits Bind Mode and restores the Channel and PN Code settings that were in use prior to entering Bind Mode. Bind Mode should last long enough for the user to locate and push the button on both the bridge and the HID. A user-initiated event can cause the HID to enter Bind Mode from any other mode.

[*The timeout value is configurable using the BIND_RETRY_COUNT define.]

2.8.4.2 Bridge

Upon entering Bind Mode the bridge sets the current channel and PN code to the channel and PN code specified in the Bind ID. The bridge listens for a Bind Request on each channel for approximately 320 ms before selecting the next channel using the Channel Selection Algorithm. This reduces the possibility of the bridge not receiving the Bind Request from the HID in the event of channel interference. If the bridge receives a Bind Request from the HID containing a supported device type, it sends a Bind Response containing the bridge's Manufacturing ID and then switches to Ping Mode. The bridge also switches to Ping Mode if the defined* time period has elapsed while in Bind Mode. The Channel Selection Algorithm uses the Bind ID to produce the channel subset for Bind Mode.

[*The timeout value is configurable using the NUM_CHANNELS_PER_SUBSET define.]

2.8.5 Data Mode

2.8.5.1 HID

When the HID application has data to send to the bridge the HID transmits a DATA packet and listens for an AutoACK. If an AutoACK is not received, the HID retransmits the packet. If the HID does not receive an AutoACK after a defined number of retransmissions of the DATA packet it assumes the channel has become unavailable due to excessive interference and moves to Reconnect Mode.

2.8.5.2 Bridge

Connected Mode allows application data to be transmitted from the HID to the bridge. The bridge should continuously listen for DATA packets from the HID. When valid data is received from the HID the bridge sends an ACK to the HID and sends the data to the USB host. If invalid data is received the bridge ignores the packet and listens for the HID to retransmit the data. The bridge monitors the interference level and moves to Ping Mode if the defined* interference threshold is reached. This ensures that the bridge is operating on a clean channel.

[*The interference threshold value is configurable using the RSSI_NOISE_THRESHOLD define]

2.9 Packet Structures

The first byte of each packet is the Header byte. Some packets may consist only of the header byte while other packets may contain up to 5 bytes.

Byte	1	
Bits:	7:4	3:0
Field:	Packet Type	Res.

Type[7:4]: The following packet types are supported:

BIND_REQ (HID) = 0x 0, // Bind Request Packet Type

BIND_RESP(bridge)= 0x0, // Bind Response Packet Type

CONNECT_REQ = 0x1, // Connect Request Packet Type

CONNECT_RESP= 0x 2, // Connect Response Packet Type

PING_PACKET = 0x 3, // Ping Packet Type

DATA_PACKET = 0x 4, // Data Packet Type

NULL_PACKET = 0x 7, // Null Packet Type

KEY_PACKET = 0x 8, // Key Packet Type for encryption

NO_PACKET = 0x F, // When there is no packet received

Res[3:0]: The lower nibble is used for packet specific information. The packet definitions below define how these four bits are used in each case.

2.9.1 Bind Request Packet (HID)

Byte	1		
Bits:	7:4	3:1	0
Field:	0	Device Type	0

Byte 1

Packet Type - 0

Device Type - This is a 3-bit field specifying a vendor-defined device type. This allows the bridge to determine what type of device the HID is and thus determine the length of data packets, which PN code to assign, etc. The keyboard device type is 2, and the mouse device type is 3.

2.9.2 Bind Response Packet (Bridge)

Byte	1				2		3		4		5
Bits:	7:4	3:1	0		7:0		7:0		7:0		7:0
Field:	0	PIN	0.		Dongle MID1		Dongle MID 2		Dongle MID 3		Dongle MID 4

Byte 1

Packet Type - 0

Pin - This is a 3-bit field specifying the PIN element of the Network ID.

Byte 2-5

Manufacturing ID (MID 1 - MID 4) – This is the 4-byte Manufacturing ID retrieved from the bridge’s radio and will be used by the HID.

2.9.3 Connect Request (HID)

Byte	1				2		3		4		5
Bits:	7:4	3:1	0		7:0		7:0		7:0		7:0
Field:	1	PIN	0.		Dongle MID1		Dongle MID 2		Dongle MID 3		Dongle MID 4

Byte 1

Device Type – 0x1

PIN – This is a 3-bit field specifying the PIN element of the Network ID.

Byte 2-5

Manufacturing ID (MID 1 - MID 4) – This is the 4-byte MID that was received from the bridge during the bind procedure. This enables the bridge to identify if the HID belongs to its network.

2.9.4 Connect Response Packet (Bridge)

Connect Response Packets are sent from the bridge to the HID in Idle and Connected Mode in response to valid Connect Requests.

Byte	1		
Bits:	7:4	3	2:0
Field:	2	Flag	0

Byte 1

Packet Type - 2

Flag (F) – This is a 1-bit field specifying a positive or negative Connect Response Packet (1 = positive, 0 = negative).

2.9.5 Ping Packet (Bridge)

Byte	1		
Bits:	7:4	3	2:0
Field:	3	Flag	0

Byte 1

Packet Type - 3

Flag (F) – This is a 1-bit field specifying a Ping or Ping Response (0 = Ping, 1 = Ping Response).

2.9.6 Data Packet (Bridge and HID)

Data packets are sent from the HID to the bridge in Connected Mode. They are also sent from the bridge to the HID in Connected Mode if there is an asynchronous back channel.

Byte	1						2		..		..		N
Bits:	7:4	3	2	1	0		7:0		7:0		7:0		7:0
Field:	4	0	Toggle	ID	0		Byte 1						Byte N

Byte 1

Packet Type - 4

Data Toggle Bit – This is a 1-bit field that is toggled for each new Data packet. It is used to distinguish between new and retransmitted packets.

Device ID – This is 1-bit field containing the least significant bit of the Device Type. The Device ID field is used in 2:1 systems to distinguish between the HID devices.

Byte 2-N

Data Byte 0-N – This is byte-aligned application data.

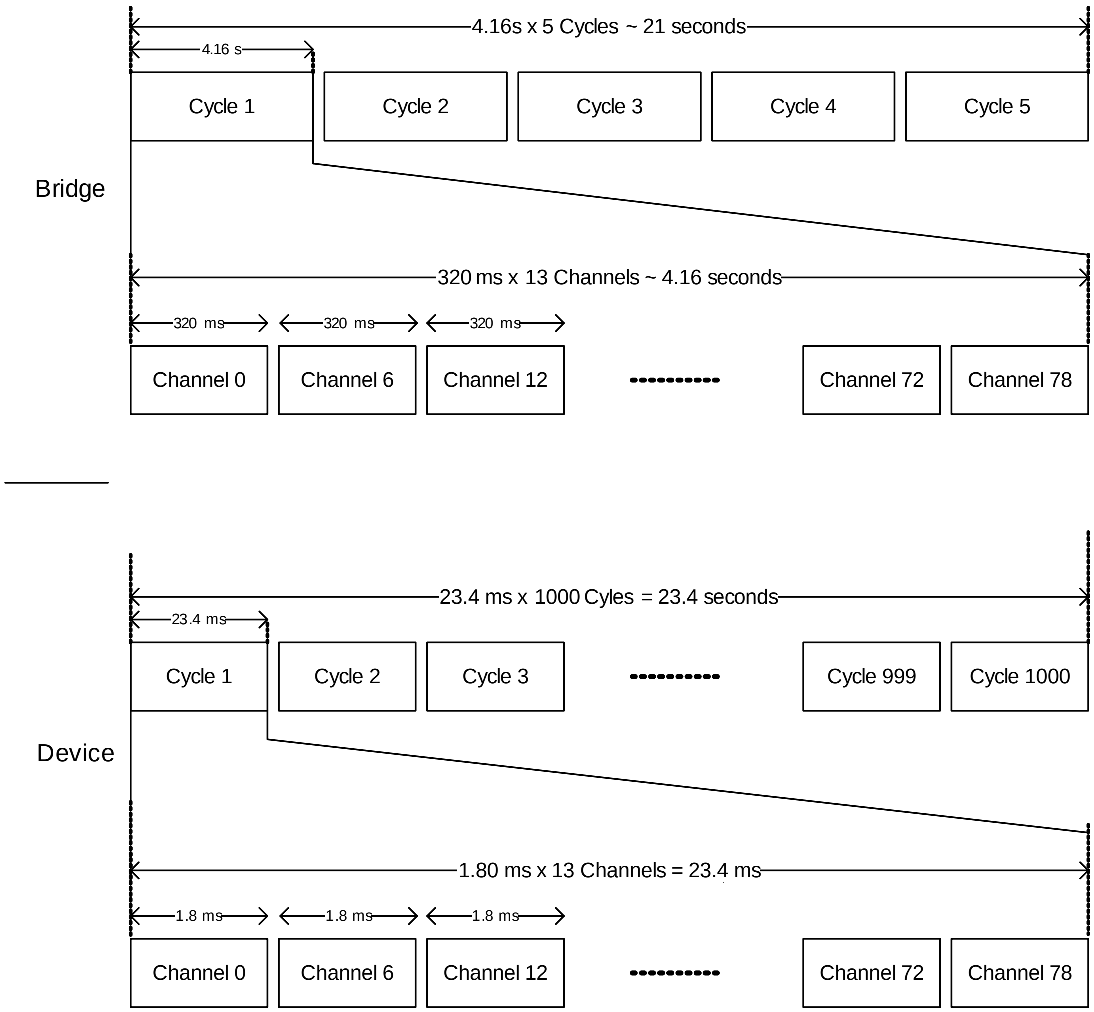
2.10 Bind and Reconnect Timing

When the Bind button on the bridge is pressed, the bridge goes into Bind Mode. In Bind Mode, the bridge uses the Bind ID to communicate with any HIDs that want to bind to the system (see section Bind Mode on page 14 for more information on the Bind ID). The bridge enables its receiver and 'listens' for any Bind Request packets from the HID, starting from channel 0. The bridge listens for approximately 320 ms on the channel, and if there's no Bind Request packet, it moves to the next channel in the Bind channel subset (the Bind channel subset consists of channels 0, 6, 12, 18, 24 … 78). It takes the bridge approximately 4.16 seconds to sequentially 'listen' on all 13 channels of the Bind channel subset. The bridge repeats the process for up to 5 times before it times out and exits Bind Mode (time out is approximately 21 seconds). If it receives a valid Bind Request packet, it immediately responds to the request with a Bind Response packet and exits the Bind Mode.

When the Bind button on the HID is pressed, the HID goes into Bind Mode. While in Bind Mode, the HID also uses the Bind ID to communicate with the bridge. The HID sends a Bind Request packet and listens for an AutoACK packet. If the HID does not receive the AutoACK, it moves to the next channel in the Bind Channel subset and repeats the Bind Request packet. It takes the HID approximately 23.4 ms to sequentially hop through all 13 channels of the Bind Channel subset, and the HID repeats the process for up to 1000 times before it times out. See Figure 2-2 on page 18.

Due to the fact that the bridge's and HID's Bind buttons may be pressed at different times, the HID and the bridge could be on very different channels when the two are in Bind Mode. However, because the HID 'hops' very quickly on all Bind channels while the bridge stays relatively long on a channel, the bridge and HID will have multiple opportunities of being on the same channel. As a result, binding normally completes very quickly as soon as the bridge and the HID are both in Bind Mode (at 1.8 ms/channel 'hopping' frequency of the HID and the bridge's 320 ms/channel, the two will 'meet' on the same channel at least 13 times in any 320 ms period).

Figure 2-2. Bind Timing Diagram

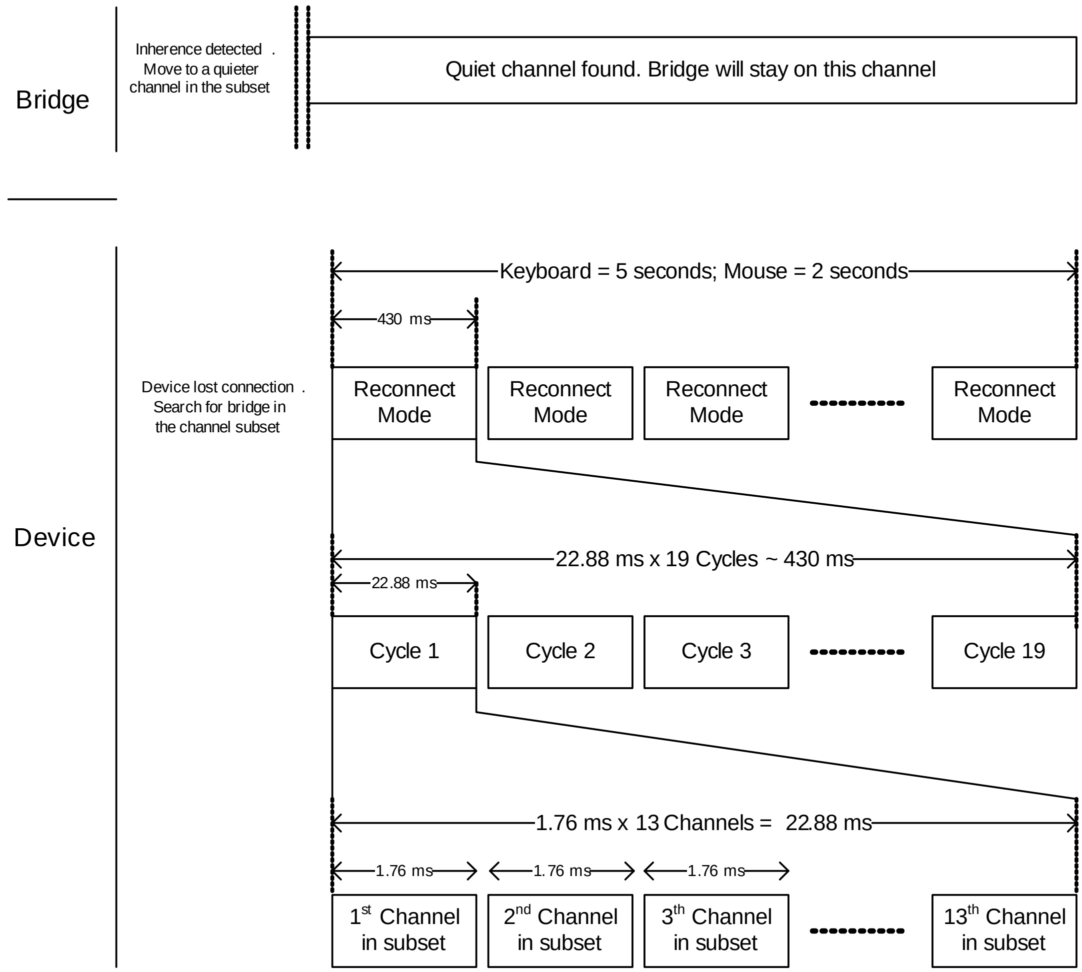


The bridge uses the Receive Signal Strength Indicator (RSSI) to determine the noise level on the channel. If the channel has become noisy, the bridge moves to Ping Mode to find a quieter channel in its channel subset.

When the HID loses connection with the bridge, it moves to Reconnect Mode to find the bridge. The HID sends a Connect Request packet and listens for an AutoACK packet. If the HID receives the AutoACK, it immediately enables its receiver and listens for the Connect Response packet from the bridge. If the HID does not receive the AutoACK, it selects the next channel using the Channel Selection Algorithm and repeats this procedure. As shown in the Figure 2-3 on page 19, the reconnect attempt takes approximately 1.76 ms/channel. The HID moves through its channel subset up to 19 times before it times out and exits Reconnect Mode. The keyboard tries to send the data for up to

5 seconds, and the mouse tries for 2 seconds, causing the HID to re-enter Reconnect Mode multiple times if necessary.

Figure 2-3. Reconnect Timing Diagram



2.11 Back Channel Support for NumLk/ScrLk/Caps Lock

In the current design, to save the keyboard power consumption, the Num Lock, Caps Lock and Scroll Lock LEDs are not supported. However, if there's a need to retrieve the LED status from the bridge to display on the keyboard, the following implementation may be used. Note that the following information is just one of the many possible implementations.

Keyboard

The keyboard may use one of the unused bits in the Data packet header to flag the bridge that it is requesting the NumLk/ScrLk/Caps Lock data from the bridge. After sending the Data packet with the flag set, the keyboard 'listens' for an AutoACK packet. If an AutoACK is not received, the keyboard retransmits the Data packet. If the keyboard does not receive an AutoACK after a defined* number of transmissions of the Data packet, it assumes the channel has become unavailable due to excessive interference and moves to Reconnect Mode. If the keyboard receives the AutoACK, it enables its receiver and starts 'listening' for data from the bridge. The keyboard will stay in this 'receive' mode for a defined* period of time. If it does not receive any bridge data, it exits 'receive' mode and tries to get the bridge data again.

Bridge

If the bridge has NumLk/ScrLk/Caps Lock data to be sent to the keyboard, it has to wait for a Data packet from the keyboard before it can send its data. If the bridge receives a Data packet from the keyboard with the flag indicating that the keyboard is requesting NumLk/ScrLk/Caps Lock data, the bridge goes into ‘transmit’ mode by enabling its transmitter to send the data. After sending the Data packet with NumLk/ScrLk/Caps Lock data, the bridge listens for an AutoACK packet. The bridge exits the transmit mode and goes back to ‘receive’ mode after it receives the AutoACK packet or after a transmission timeout.

Both the keyboard and the bridge must keep track of the Data Toggle bit for the ‘bridge → keyboard’ data transmission. The Data Toggle bit is reset to ‘0’ after a successful Bind. The Data Toggle maintained in the keyboard is toggled when the keyboard receives a valid Data packet from the bridge. The Data Toggle maintained in the bridge is toggled when the bridge receives the AutoACK from the keyboard. If the keyboard receives a mismatched Data Toggle bit from the bridge, it does not update its Data Toggle and ignores the received Data packet. This Data Toggle is independent of the ‘keyboard → bridge’ Data transmission.

[*The timeout is application specific and should be defined by the developer]

2.12 Signature Byte

The WirelessUSB LP RDK uses the SIGNATURE byte to determine if the HID has ever been bound to any bridge before.

If the HID has never bound to a bridge, the non-volatile memory used to store the SIGNATURE and the bridge's MID data remains in its default value. Once the HID has bound to a bridge, the SIGNATURE byte is set to 0x90 and the bridge's MID is also stored.

At power up, the HID reads the SIGNATURE and the MID bytes to determine its next action. If the SIGNATURE byte is 0x90, the HID uses the retrieved MID to calculate the NetworkID and moves to Re-connect mode. If the SIGNATURE byte is not 0x90, the HID goes to sleep mode, waiting for the user to initiate the Bind process.

2.13 Encryption

Data packets may be encrypted for privacy. All encrypted Data packets shall have a payload of 8 bytes; this is the minimum block size for the encryption algorithm.

WirelessUSB LP uses the Tiny Encryption Algorithm (TEA) to encrypt application data. Some of the features of TEA are:

- 128-bit encryption key
- 8-byte block size
- Minimal RAM requirements
- Small code size
- Highly resistant to differential cryptanalysis

In order to use the TEA algorithm both the bridge and HIDs must possess the Data Encryption Key. The bridge is responsible for creating the key, which is then shared with the HIDs. There are a variety of possible methods to share the key between the two devices. The key may be exchanged over the WirelessUSB link using the Encryption Key Request and Encryption Key Response packets.

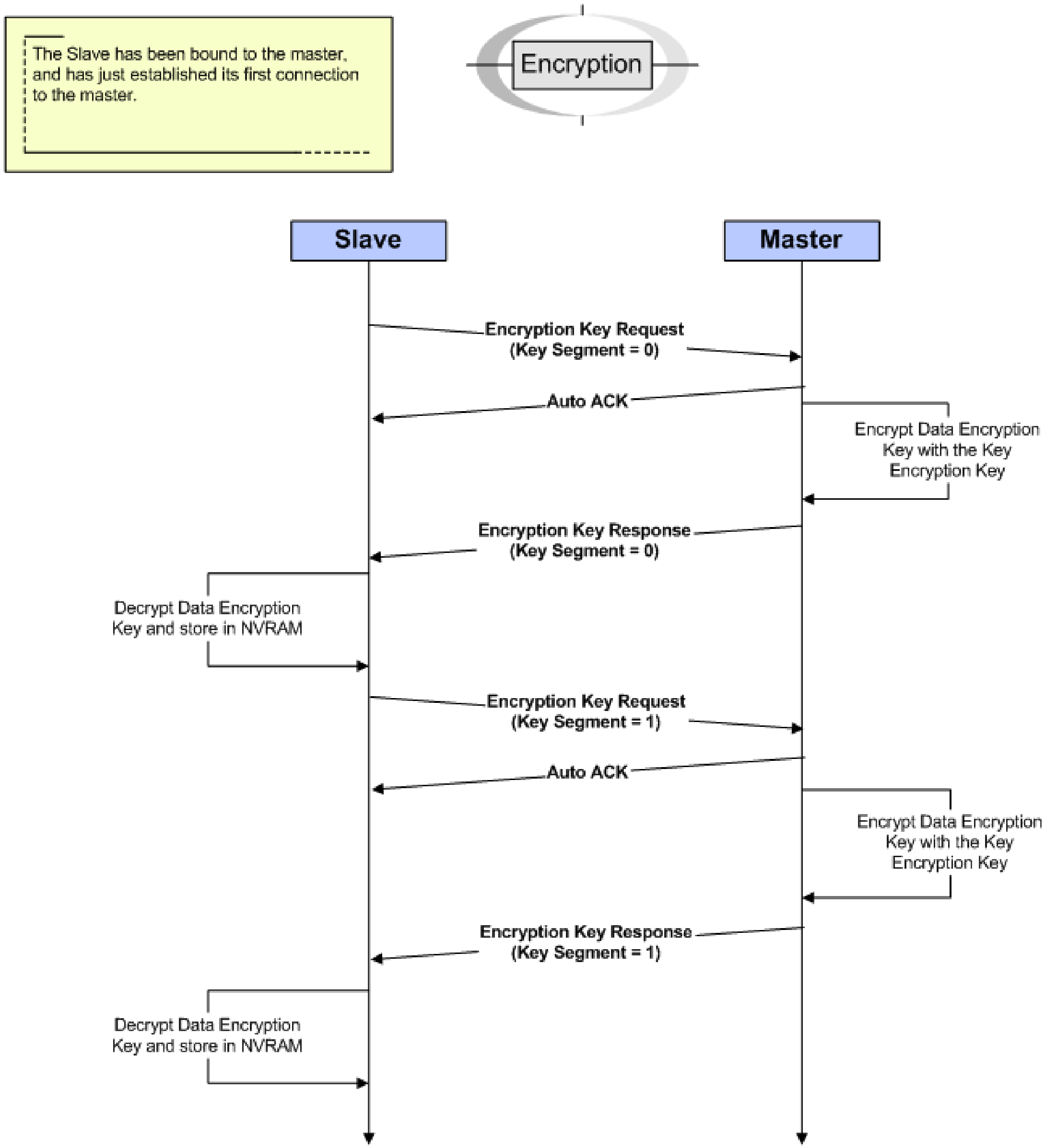
2.13.1 Key Management Over WirelessUSB

After binding and connecting to the bridge, the HID transmits an Encryption Key Request Packet and listens for an AutoACK followed by an Encryption Key Response Packet containing the first half of the Data Encryption Key. The HID then uses the Key Encryption Key (calculated from the bridge and the HID MIDs) to decrypt the Data Encryption Key. The HID repeats this process for the second half of the Data Encryption Key and stores the key in NVRAM. After receiving both halves of the Data Encryption Key the HID may begin transmitting encrypted data to the bridge.

2.13.2 Encryption and Power Consumption Trade Off

If the keyboard encryption is enabled, each key code is encrypted into an 8 byte key code (the Keep Alive and key up packets are not encrypted). When a single key is pressed, a non-encrypted key down packet consists of 16-bit Preamble + 2 bytes SOF + 1 byte packet header + 1 byte key code + 2 bytes CRC while an encrypted key down packet consists of the same overhead packets plus 8 byte key code instead of 1 byte key code. As a result, the active transmit Icc increases by approximately 50%. This results in an increase in the average power consumption when encryption is enabled. Our measurement shows that the average Icc in test mode (sending “quick brown …” string) is 3.33mA if encryption is disabled. If encryption is enabled, the average Icc increases to 3.66mA.

Figure 2-4. Encryption Key Management



3. Mouse

3.1 Introduction

3.1.1 Overview

This section describes the design goals, architecture, firmware source code modules and configuration options for the WirelessUSB PRoC LP mouse. It does not cover the details of the radio subsystem or the configuration options that go with it.

3.1.2 Design Features

The CY4672 Reference Design Kit uses a low cost Programmable Radio on Chip-LP for the RDK mouse (Cypress part #: CYRF69103-40LFXC). Contact your local sales representative for more information on the part.

The architecture was designed to be modular for extendibility and maintainability. It was also designed so that it could easily be ported from one hardware platform to another assuming the use of an equivalent microprocessor. Porting to another microprocessor family requires more work to account for hardware specific changes.

Design efforts have been made to reduce the 'on' time of the microprocessor and radio to conserve battery life. This includes protocol optimizations along with using sleep features of the radio, mouse optics and the microprocessor.

3.2 Hardware Overview

This section presents the RDK mouse assembly, the hardware block diagram, schematics, and Hardware Considerations.

3.2.1 RDK Mouse Assembly

The RDK mouse is currently enclosed in a skin that has been designed for the Avago ADNS-3040 Ultra Low-Power mouse sensor. The mouse features three buttons with one button combined with the scroll wheel function. There is a connect button on the bottom of the mouse allowing the user to perform an explicit bind with the bridge.

Figure 3-1. Bottom View Bind Button and On-Off Switch

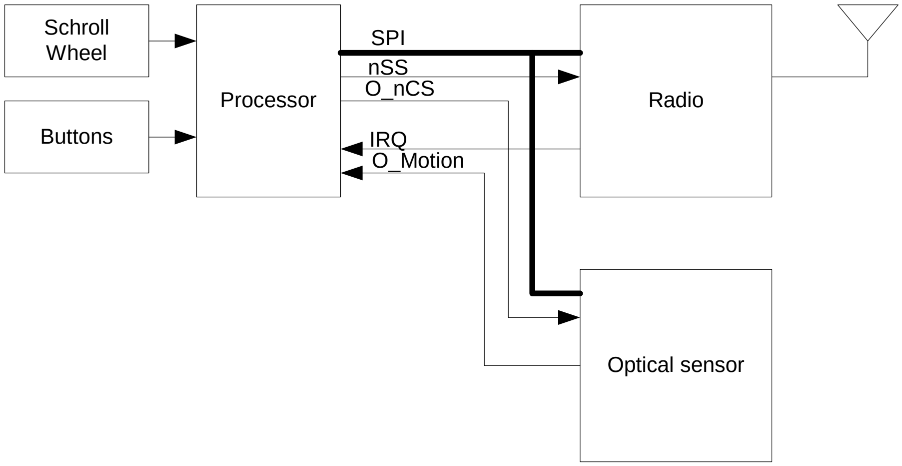
Figure 3-1 shows the bottom of the mouse with the optics window, power switch, and Bind button. There are two screw holes above the label. The top of the mouse can be removed once these two screws on the bottom and one screw on the top have been extracted.

Figure 3-2. Exploded Mouse View

Figure 3-2 is a picture of the mouse with the top removed. The mouse consists of a single PCB that contains all of the necessary mouse components.

3.2.2 Hardware Block Diagram

Figure 3-3. Mouse Hardware Block Diagram



3.2.3 Schematics

All schematics for the optical wireless mouse are located in the following directory: <installation directory>\Hardware\RDK mouse. The schematic is in Adobe Acrobat format with the letters 'Sch' in the file name.

Figure 3-4. Printed Circuit Assembly (PDC-9302)

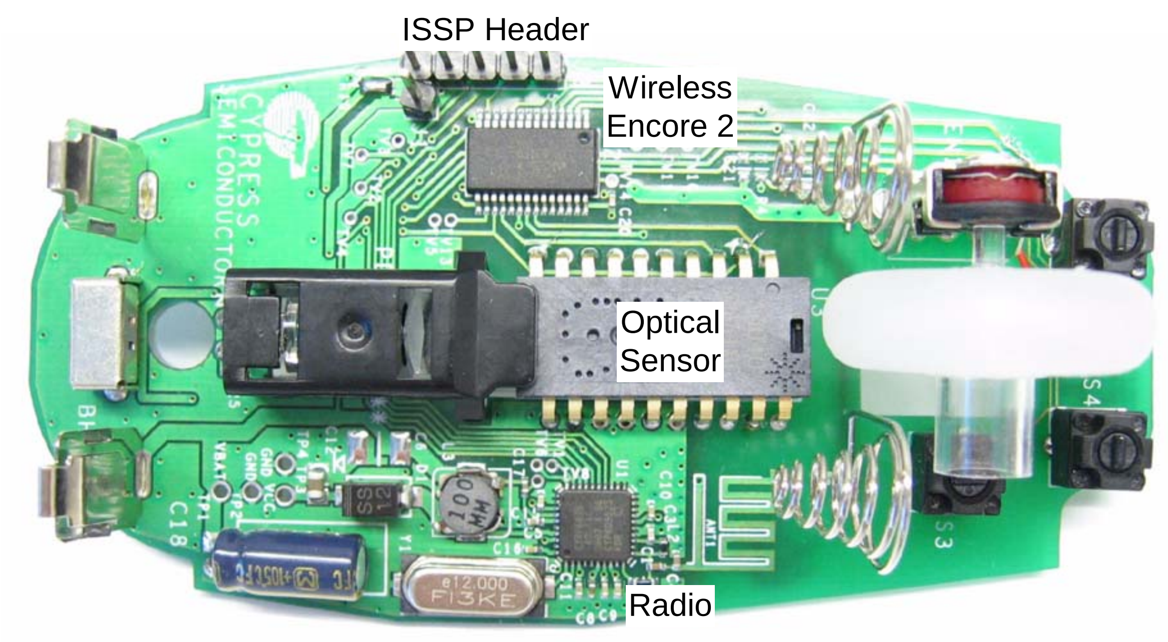


Figure 3-4 is a picture of the controller board with the WirelessUSB PRoC LP Radio and optical sensor. The 'wiggle' trace in the lower right is the antenna. This board has the option of adding pull up resistors and filtering capacitors to the z-wheel and then powering the z-wheel with a separate GPIO pin on the microcontroller. J10 is a programming header. Either the ICE-Cube or the PSoC MiniProg may be used to program the mouse microcontroller using this ISSP header. J10 also doubles as a mechanism to force the firmware to enter a manufacturing test mode that is compatible with the Cypress Manufacturing Test Kit. Pins 4 and 5 must be shorted together using a shorting block before power is applied to the mouse to enter this mode.

3.2.4 Hardware Considerations

The mouse design uses the SS12 schottky diode (D1) and CDH53100LC inductor (L3) for its boost circuitry. With these high efficiency components, preliminary characterization data shows a range of approximately 74-87% efficiency for the 1.8-2.7V VBAT voltage range at different temperatures (-10C to 80C). The mouse is a higher power consumption device compared to the keyboard. Extending the battery life is one of the crucial design considerations in the mouse design. The trade off for a higher efficiency boost circuitry is the component costs and the board size (these components are slightly bigger in size compared to the ones used in the keyboard design).

3.3 Firmware Architecture

There are two architectural views of the mouse. The first is a microcontroller configuration view of User Modules mapped to digital and analog blocks inside the microcontroller. This architecture and configuration is best viewed in the PSoC^{TM} Designer application when the project is loaded. The second view is a logical organization of the source code modules that make up the mouse application code and other support modules.

This section describes both architectures with emphasis on top level organization and overall module operation. More detailed description of variables and functions should be obtained by studying the source code.

3.3.1 ROM/RAM Usage

The following table shows the ROM/RAM usage. The top part exhibits the total ROM/RAM usage for basic functions, which disables all the build options below. The bottom part exhibits the ROM/RAM usage for individual build options.

Table 3-1. ROM/RAM Usage

	Total ROM (Bytes)	Total RAM (Bytes)
Basic Function	5202	69

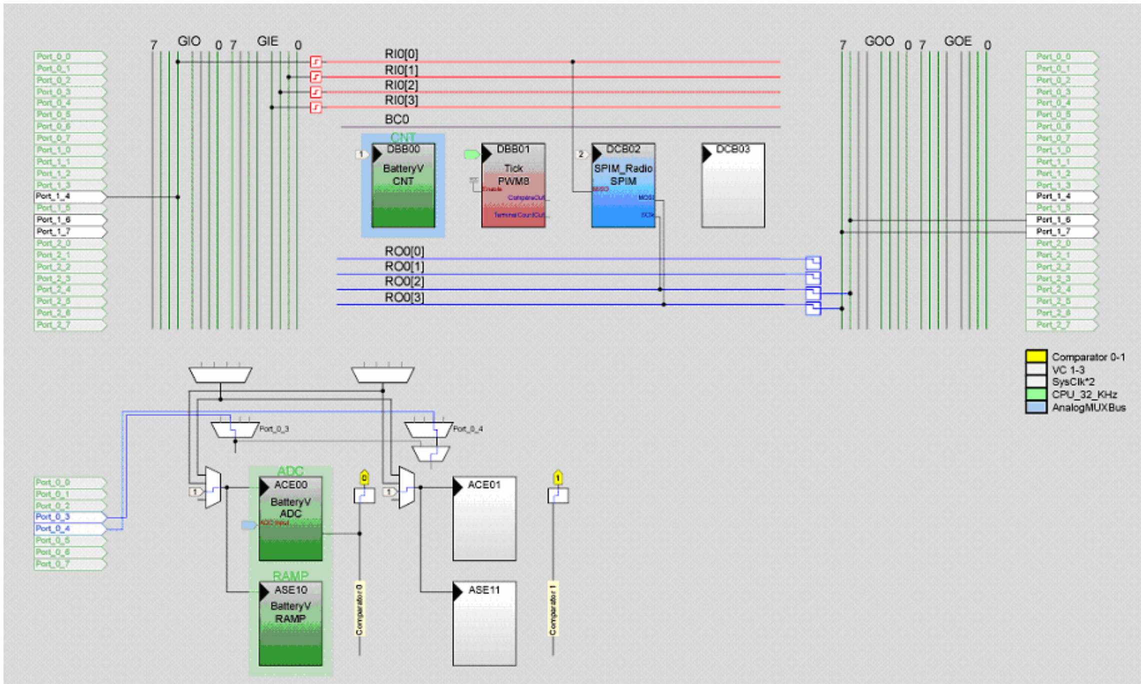
Build Option	ROM Usage (Bytes)	RAM Usage (Bytes)
MOUSE_BATTERY_STATUS	708	2
MOUSE_TEST_CODE	522	0
MFG_TEST_CODE	448	0
MFG_TX_MODES	735	3

3.3.2 enCoRe™ III LV Device Configuration

The enCoRe III LV is configured using the Device Editor in PSoC Designer. The mouse uses three digital blocks and two analog blocks to support three separate user modules. The first module is an SPI master for communicating with the optical sensor and the radio. The second module is a PWM configured to operate as an 8-bit timer. The third module is a 10-bit ADC used to measure the battery voltage. The ADC is the module that requires two analog blocks in addition to one digital block.

Figure 3-5 is a block diagram of the device User Module mapping to digital and analog blocks. Further description of resources and User Modules follow the diagram.

Figure 3-5. Microcontroller Device Architecture



3.3.2.1 *Global Configuration*

Following is a description of the **Global Resources** that are configured for the CY7C60323-PVXC enCoRe III LV microcontroller. Care must be taken when modifying these values as they affect the User Modules discussed below.

3.3.2.1.1 Power Setting / System Clock Frequency

The mouse is powered using the Power Management Unit on the CYRF6936 radio chip. The default configuration of the radio PMU is used to power the microcontroller at 2.7 volts. The microcontroller power setting is set to 2.7 volts / 12 MHz.

3.3.2.1.2 CPU Clock

This parameter is set to SysClk/1 in order to run the CPU at 12 MHz. This operating frequency provides for faster code execution, when events are detected, so the microcontroller can be put back into sleep mode for improved power savings. Slower clock frequencies increase power consumption of the microcontroller in the mouse design.

3.3.2.1.3 Sleep Timer

The sleep timer is set to 1 Hz in order to reduce the microcontroller load. The sleep timer is used to maintain a very coarse sense of time when the PWM User Module is turned off. Changing this value affects timing operation of the **timer** code module.

3.3.2.1.4 VC1 – VC3

The other clock sources are set to provide optimal performance for the SPI User Module and the ADC. Since there is a dependency of VC3 on VC2, VC2 on VC1 and VC1 on System Clock, changing of these parameters have a cascading affect. VC1 is configured for SysClk / 2. VC2 is configured for VC1/3 and is used to clock the SPI User Module with 2 MHz. VC3 is used to clock the ADC User Module and is configured for N = 128. The clock source for VC3 is set for VC1.

3.3.2.1.5 Other Global Resources

The SysClk Source must be set to Internal 24 MHz and the SysClk*2 Disable must be set to Yes. The Trip Voltage is typically set to the lowest value of 2.45 V. LVD ThrottleBack must be disabled. The Watchdog Timer is typically enabled.

3.3.2.2 SPI Master User Module

The SPI Master User Module is used to communicate with both the radio and the optical sensor. Both devices support leading edge data latching, non-inverted clock, and MSB first transmission as defaults. This module uses the VC2 clock resource of 2 MHz which is divided by two for a 1 MHz SPI clock. For reliable operation it is not recommended to increase the SPI clock frequency. The interrupt API to this module is not used. See the **spi** code module description for how this module is used to implement communication with multiple devices on the SPI bus.

3.3.2.3 PWM User Module

The PWM User Module is configured to use the ILO oscillator (CPU 32-kHz clock). This module is used to provide a periodic interrupt to the **timer** code module in order to maintain a power saving millisecond sleep routine. The period of the timer is calibrated to the system clock at power on in order to provide a period of about 250 µs. This calibration is performed to account for variations in temperature and ILO variances from part to part. The module must be configured to generate a terminal count interrupt. The period parameter is ignored since it is programmed at run time based upon the calibration results. See the **timer** code module for more details on calibration.

3.3.2.4 ADC User Module

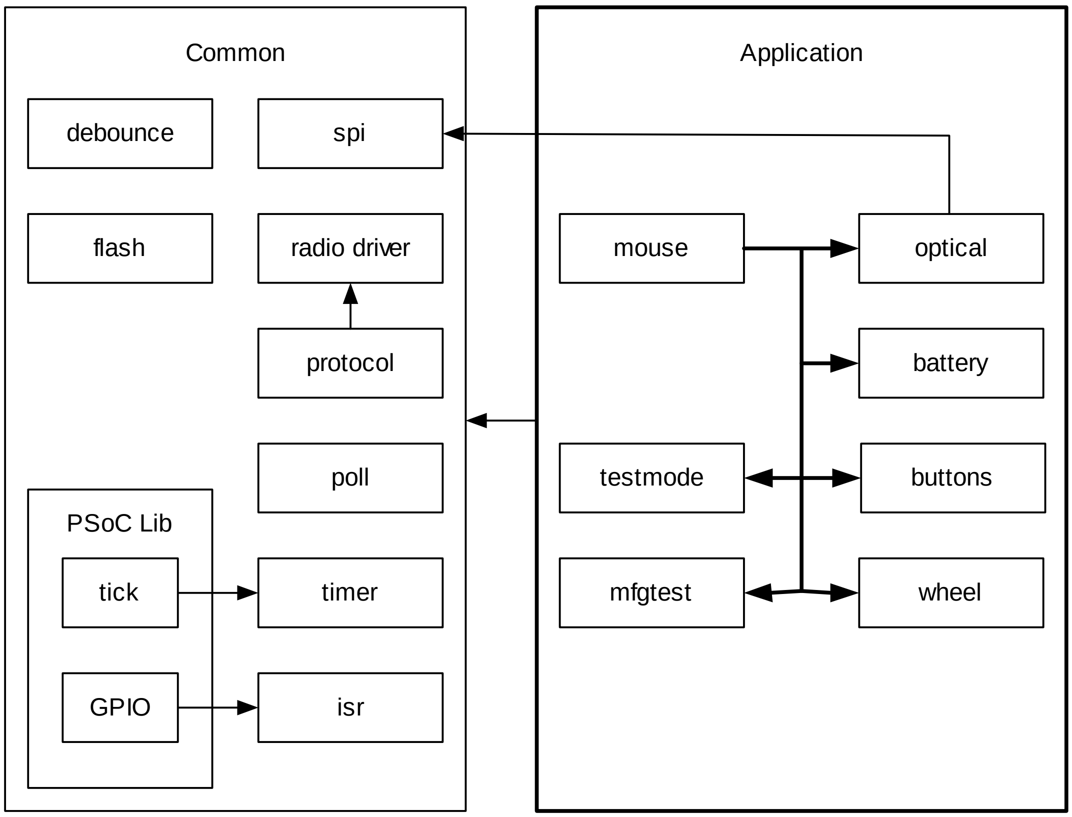
The ADC User Module is only powered on when needed to make a battery voltage measurement. The VC1/VC3 global clock resources are used to control the measurement timing. The PWM High parameter must be set to 8 VC3 periods and the PWM Low parameter must be set to 1 VC3 periods. The interrupt API must be enabled for the module to make a measurement. The battery voltage is filtered and then connected to a microcontroller pin that is routed to the AnalogMUXBus, which in turn is routed to the ADC.

3.3.2.5 Flash Security

The PSoC Designer mouse project has a file called *FlashSecurity.txt*. This file specifies access rules to blocks of the Flash ROM. Refer to the documentation at the top of the file for definitions. This file is shipped with a single change from its default configuration. The block starting at hex address 1FC0 has been changed from W: Full (Write protected) to U: Unprotected. This location of Flash has been dedicated to saving non-volatile configuration values for the **protocol** code module. Note: when building the mouse firmware, be sure to check that the text image size does not occupy this block.

3.3.3 Model

Figure 3-6. Firmware Architecture Model



The mouse firmware is partitioned into two logical groups. The Common group is a collection of code modules that provide the underlying support for the application. This group provides services such as radio protocol, radio driver, timing, polling, flash access, contact debounce, SPI, and interrupts.

The Application group implements the core functionality and features of the RDK wireless mouse. This includes power management, optical sensor, button, z-wheel, packet formatting and reporting, various test modes and battery level sensing. The code modules for each of these groups are described below in further detail.

All of the following module descriptions have corresponding <module name>.c or <module name>.asm and <module name>.h source code files. The module API and definitions are exported in the header file while the module implementation and local definitions are contained in the C/ assembly file.

3.3.4 Common Code

The modules in the common code group are a combination of two sources. The first is PSoC Designer generated files in the *lib* directory that have been modified to support the application. The second group is modules that are generally used by the application.

3.3.4.1 Generated Library Code

There are currently only two files, generated by PSoC Designer, that are modified for the use of the application. A minimal amount of code has been added to these modules in user protected areas that are preserved across code generation.

3.3.4.1.1 GPIO Interrupt Module

Due to inefficiency in the handling of compiler defined registers at interrupt level, code was added to this module to improve the efficiency of interrupt handling in C modules to maintain modularity. Refer to the **isr** module for more details.

3.3.4.1.2 Timer Interrupt Module

The timer interrupt module has been modified to provide a finer timing of 250 µs for the **poll** module and course timing by providing a 1 ms tick. When the timer module has been turned off, it still provides a sense of time on the 1 ms tick by using the sleep timer. In this case polling is disabled to conserve power. See the **poll** module and **timer** module for more details.

3.3.4.2 Debounce Module

The debounce module is an assembly coded routine to perform debounce on button presses as well as z-wheel motion. The algorithm is one that was published in EDN article as a way to perform hardware debounce in software.

The debounce is performed by polling the inputs at a fixed period and by adding a weighted value of the input to an accumulated value carried from the previous poll. The output is then passed to threshold logic, with built in hysteresis, and a logic value of one or zero is computed. The thresholds can be changed to adjust the hysteresis crossings by setting SCHMITT_HIGH_THRESH and SCHMITT_LOW_THRESH. Once an input has changed state the output can be observed to change approximately 10x the poll period later with the current threshold settings. With a poll period of 250 µs the input latency would be about 2.5 ms.

Refer to *Contact-debouncing algorithm emulates Schmitt trigger* on http://www.edn.com for more details on the operation of this algorithm.

3.3.4.3 SPI Module

This module provides an interface to the SPI bus for the optical sensor only. Physically the SPI bus is connected to the radio and the optical sensor. The radio driver is responsible for interfacing with the radio. The enCoRe III LV SPI Master module does not manage the selection of slave devices. This module was created to provide that functionality. This module has a dependency on the instantiation of a SPIM module in PSoC Designer that is properly connected to the devices.

3.3.4.4 Radio Driver

The radio driver module is a low level module providing basic radio communication and configuration. Its general application is such that it is likely not to be changed by the firmware developer. It provides an interface for reading/writing radio registers, setting PN codes and initialization of the radio and transmitting or receiving packets. See the WirelessUSB LP Radio Driver documentation for details.

3.3.4.5 Protocol Module

The protocol module defines and implements the layer used to deliver packets from the device to the bridge. It manages the binding of devices to a bridge as well as the connection and interference immunity by channel hopping. This module has a dependency on the **radio driver** for sending formatted packets and the **flash** module for storing the manufacturing ID of the bridge the device is bound to.

3.3.4.6 Flash Module

The flash module is a smaller version of E2PROM module provided in PSoC Designer. It is limited in functionality and only implements the read/write routines required by the device. The *flashsecurity.txt* file must be modified so that the block being modified by this module is given read/write privilege, i.e. unprotected. Currently the very top most block in flash is used for this module.

3.3.4.7 Port Module

GPIO pins on the enCoRe III LV microcontroller ports can be configured as outputs with a pull up resistor. This is the case for mouse buttons and the Bind button. In order to activate the pull up a data value of one must be written to the port data latch for the pin. This feature presents a problem when performing a read-modify-write on the port. For example, if a button happened to be pressed (grounding the pin) a zero would be read and written back out on the read-modify-write operation. This would turn off the pull-up for the button thereby essentially disabling the button. The port module provides an interface to treat ports, using the pull up feature, in a special way by caching the drive data for the port.

3.3.4.8 Poll Module

The poll module manages the timing, enabling/disabling and polling of the mouse buttons and z-wheel inputs. When the mouse is active, polling is enabled and occurs at a rate of about 250 µs for the z-wheel (see the **timer** module) and a rate of about 3 ms for the mouse buttons. When the mouse is inactive, the buttons are changed to interrupt mode and the z-wheel is polled for change only when the sleep timer expires; see **button/wheel** modules.

3.3.4.9 Timer Module

The enCoRe III LV has an internal low power oscillator (ILO) that is used for generating a clock to a PWM User Module. This clock is affected by voltage and temperature and may drift over time. This module provides an interface to periodically calibrate the PWM to the system clock. The PWM period is calibrated to be approximately 250 µs. Care must be taken when changing this period since the **poll/debounce** modules are coupled to this time value. Calibration of the ILO is done by placing it in a single power mode. Since the ILO configuration register is read-only, the fastest frequency is set for a more precise calibration. Typical frequencies tend to be in the 90-100 kHz range. Because the ILO is adjusted this way, it also affects the sleep timer. Setting the sleep timer to 1 second in the device editor actually results in a sleep timeout on the order of ¼ second.

The timer module also provides a set of functions for performing busy waits in the microsecond resolution. For more coarse timing requirements an API is provided for millisecond delays. The millisecond delay routines should be used as often as possible to provide for better power consumption since the microcontroller sleep feature is used. Also, when polling is enabled, it is performed as a background task during the millisecond delay.

This module also adjusts the tick advancement based upon the sleep resolution selected as a global parameter in the Device Editor when the timer is disabled. Turning off the timer provides for more power savings, yet a sense of time is still preserved for non-critical timing.

Note: When using the ICE-Cube, define the macro PSOC_ICE so that busy waits are used instead of the sleep instruction. Using the sleep instruction with the ICE-Cube generates errors due to synchronization issues between the OCD part and the emulator.

3.3.4.10 ISR Module

This module provides an interface to enable/disable individual GPIO interrupts. It also provides the top level GPIO interrupt handler for the application. It does not handle User Module specific interrupts; those are handled by specific routines provided in each module.

Due to compiler inefficiencies and the desire to maintain modularity with interrupt handlers, the top level handler *isr_gpio*() calls a routine in the GPIO interrupt User Module. The parameter passed is a pointer to another C function that has been declared as an interrupt handler. This is done in order to preserve the registers used in the handler. The *gpio_isr_redirector*() function manipulates the stack, preparing it for the interrupt handler being called. The rule is that the module routine being called may not call any other functions and it must be declared with the #pragma interrupt_handler. For example, the interrupt handler in the **optical** module to handle detection of optical motion does not call any other C functions. Instead, it uses a macro provided by the mouse module to post an event flag. See the **optical/mouse** modules. This method prevents the compiler from pushing all of its virtual registers onto the stack when a function is called at interrupt context.

3.3.5 Application Code

The group of modules that make up the application code are responsible for implementing the mouse functionality and behavior. Following is a high level description of each module responsibility and associated algorithms.

3.3.5.1 Mouse Module

The mouse module is the controlling code for the application. It has many responsibilities in implementing various features and functions offered by the mouse. The data formats and reporting algorithms along with power management are explained in this section.

A few types are defined to support the operation of the mouse. One of these is the packet format used when sending data to the bridge. This type is defined as TX_PACKET and is structured to support the different data packet formats as explained in the section Wireless Protocol Data Payload on page 37. The present definition combines z-wheel data with button data into one byte in order to conserve battery power by shortening the on-time of the radio. This format would need to change in order to support a mouse with more than three buttons and a z-wheel, perhaps sending four bytes instead of three.

The function *main*() is the entry point for the mouse application. This function is called from the *boot.asm* file. The mouse first initializes all of the application modules and then initializes the **protocol** module. There is an order dependency for some of these, so care must be taken in modifying the *mouse_init*() function. For example, other modules depend upon the timer facility running in order to perform initialization. The **spi** module must be initialized before the **optical** and **protocol** modules can be initialized. Once each module has been initialized, then the application checks for entry to the 'LP' draw test mode or the manufacturing test mode. If neither of the test modes is indicated, then normal mouse operation begins.

The mouse module handles a variety of events at the main thread level. Most interrupt routines post notification that an event occurred by using the macros provided by the mouse interface. The mouse then processes these events at thread context rather than interrupt context.

The mouse application is implemented using a state machine to manage the various power modes that it executes at any given time.

The mouse initially enters a disconnected state. When there is any mouse activity, it enters the active state.

In the active state the timer is turned on so that more accurate timing and mouse events can be collected, formatted and reported to the bridge. The mouse remains in this state as long as there is mouse activity to report to the bridge or a period of time without any mouse activity has expired, after which it returns to the idle state. If the mouse is unable to deliver a packet while in this state, it transitions to the disconnected state.

In idle state the optical sensor is allowed to transition through its various rest modes to conserve power. In this state, the mouse application is waiting for input from the optical sensor, z-wheel or buttons. The timer is turned off to conserve power and the notion of time is maintained using the sleep timer. This state is maintained indefinitely until the batteries drop below 1.8 volts at which point the mouse enters the off state.

The off state is where the radio and optical sensor are prevented from turning on. This state is reached when the battery voltage drops below 1.8 volts. It is designed to keep the battery drain to an absolute minimum to prevent battery leakage as a result of completely draining the batteries.

The battery level is reported by the mouse application when it detects a change from the disconnected state to the connected state. The battery level is measured when exiting the idle state. If there is a change in the battery level, then it will be reported in the active state.

In the active state the mouse attempts to deliver a packet for the amount of time designated in MOUSE_TX_TIMEOUT_MS. If it is unable to send the packet in this time, then it transitions to the disconnected state.

The mouse application is responsible for detecting the Bind button press and then calling the bind function in the **protocol** module.

The mouse application sends mouse reports as frequently as events arrive, but not any faster than the time defined in the macro MOUSE_REPORT_IN_MS. Care must be used when setting this time so that the report rate does not exceed that which the USB bus is capable of handling. Keep in mind that the report rate varies slightly due to drift of the internal oscillator used to keep track of time.

3.3.5.2 *Optical Module*

The optical sensor module encapsulates the initialization, calibration and reading of the optical sensor. This module also handles any power management required by the sensor, along with motion detection if supported. The contents of this module potentially change with every design and are unique to the sensor used.

This module has the responsibility to format the X and Y data into the mouse packet payload. Refer to section Wireless Protocol Data Payload on page 37 for a definition of the packet payload.

3.3.5.3 *Battery Module*

The battery module is responsible for measuring the battery voltage and converting it into a level between one and ten using an A/D.

The battery monitor circuit is implemented using a resistor and capacitor to filter out ripple from the switcher, and is routed to an analog input on the enCoRe III LV microcontroller. This input is then connected to a 10 bit single slope analog to digital converter User Module. The firmware has been implemented to read the battery voltage and then provide a ten level mapping of the battery voltage. The battery level is periodically measured and changes in level are transmitted to the bridge; see the **mouse** module for the algorithm.

The battery module must be calibrated at power on reset prior to use in order to compensate for part-to-part differences. Calibration is accomplished by calling:

BatteryV_iCal(BATT_CAL_SET_POINT, BatteryV_CAL_VBG)

prior to each use. This forces the A/D to calibrate as close as possible to 1.3 V. BATT_CAL_SET_POINT is = 475 (1.3V /2.8 V * 1023) which is the digital representation of 1.3 volts using the internal band gap reference.

3.3.5.4 *Testmode Module*

The Testmode module provides code to continuously perform a vector drawing test within a drawing application. This test mode is used to check radio range, co-location and inter-operability of the mouse with the keyboard.

The test mode, when compiled in, is entered by holding down the left and right button while inserting the batteries. The buttons must be held down until the optical sensor begins to flash. As soon as the buttons are released the mouse repeatedly draws ‘LP’ in the drawing application. Each successive ‘LP’ should be drawn on top of the previous one. The test mode may only be exited by removing the batteries. All button presses and mouse movement are ignored when in the test mode. However, care must be taken not to bump other mice connected to the PC.

Note the mouse ‘acceleration’ or ‘enhance pointer precision’ option must to be turned off in the Windows mouse Control Panel for this test to execute properly. If the letters are drawn erratically with uneven sides or excessive amounts of space in between them, then check this setting or its equivalent (based upon your PC operating system).

When the macro DEBUG_INDEX is defined, code is generated to move the mouse pointer to the right and back again without the pen down. This is done in an incrementing fashion so that when observing packet data on a Listener, a correlation can be made with a USB protocol analyzer. This is useful for debugging data loss since the test mode guarantees packet delivery.

Entry to this test mode can be changed by modifying the macro TESTMODE_BUTTONS in the *testmode.c* file. The button macros are defined in the *buttons.h* file.

3.3.5.5 Buttons Module

The buttons module provides an API for handling the Bind button and the mouse buttons. This module must be changed when adding or removing buttons for a new mouse design. The button portion of the packet payload is formatted by this module and will need to change if more buttons are added. See the **mouse** module for a definition of the packet payload format.

This module manages power configurations that may be implemented to conserve power related to button presses. For example, button polling is turned off and interrupts are used to detect button presses in the idle state. It also manages the acquisition of button information depending on the implementation: interrupts or polling.

When changes in button state are detected, the **mouse** module is notified for collection and reporting of the data. Note: it is important for the buttons module to always report the button state when a button is pressed. This condition frequently occurs when the mouse is moved with the button held down.

3.3.5.6 Mfgtest Module

The manufacturing test module may be optionally compiled in, at the expense of code space, by defining the macro MFG_TEST_CODE. In addition, a more complete version may be compiled in by defining MFG_TX_MODES. The TX modes include code to perform a carrier test as well as a random data test.

The manufacturing test code is designed to be compatible with the CY3631 Manufacturing Test Kit Tester. Entry into this mode on the mouse is performed by placing a shorting block over pins four and five of the ISSP programming header and then inserting the batteries. The test mode may only be exited by removing the batteries and shorting block. For more information on how to use this test mode, refer to the CY3631 Manufacturing Test Kit documentation.

It is recommended that you not make changes to this module unless similar changes are made to the CY3631 Tester.

3.3.5.7 Wheel Module

The wheel module implements the functionality of the z-wheel. It is responsible for power modes associated with the z-wheel, polling, z-wheel interrupts, wheel position tracking, and partial packet formatting for z-wheel reports.

When the z-wheel is being polled, the GPIO pins are turned on with internal pull up resistors just long enough to read the state. This is done to conserve power when the mouse is active. When the polling timer has been turned off the *wheel_poll_sleep*() function is called which only looks for change from the last state; it does not keep track of wheel position.

Z-wheel position tracking is done by comparing debounced wheel input to the previous two states. Depending upon the wheel input phase transition the direction of the wheel can be determined. The poll rate must be frequent enough to debounce and catch these transitions for a smooth response. The RDK mouse is shipped with a mechanical encoder. It is typical for this decoder to rest on a detent such that the z-wheel inputs are either both high or both low, hence the reason for only turning on the pull ups when polling the input. Transition from one of these states to the other is reported as a +/-1 motion. Note: sometimes the mechanical detents don't align with the high-high or low-low state and movement may not be seen every time from detent to detent.

When z-wheel motion is detected, the **mouse** module is notified for collection and reporting of the data.

3.3.6 Configuration Options

All configuration options for the application can be found in the *config.h* file. Each item is explained below and can be changed to values that meet the developer's requirements.

3.3.6.1 MOUSE_REPORT_IN_MS

This configuration value sets the shortest period at which the firmware will honor events from the mouse hardware to transmit using the radio. The default value is approximately 10 milliseconds. Setting this value to something smaller than the USB poll period of 8 milliseconds generates excessive radio retries from the mouse and is not recommended. Larger values improve battery life, but may affect usability of the mouse. See the **timer** module for a description of timing accuracy. This valued is defined in milliseconds.

3.3.6.2 MOUSE_ACTIVE_MS

This value sets how long the timer module runs generating poll interrupts for the z-wheel and buttons. This time affects power consumption of the mouse. Once this time expires, the buttons and z-wheel go into a power down state, improving battery life. In power down state, z-wheel movement exhibits latency. See the **button/wheel** modules for descriptions of power down states and operation. This value is defined in milliseconds.

3.3.6.3 MOUSE_DISCONNECTED_POLL_MS

Sets the rate at which the battery voltage is monitored while in the disconnected state. This ensures that if the batteries go below the minimum battery voltage of 1.8 V, the radio and optical sensor are prevented from turning on.

3.3.6.4 MOUSE_TX_TIMEOUT_MS

The transmit loop in the mouse attempts to guarantee delivery of mouse events. This loop eventually times out if it does not receive a response from the bridge. This value sets that time-out time. The default value is 2000. This value is defined in milliseconds.

3.3.6.5 PLATFORM_H

This configuration value identifies the header file that has the platform configuration information. The default value is *pdc9302.h*, which is the identifier for the mouse board that is shipped with the RDK. This macro changes when the code is ported to another platform.

3.3.6.6 *MOUSE_800_NOT_400_CPI*

This configuration definition is used to select between 800 or 400 counts per inch (cpi) when configuring the optical chip. If it is defined then 800 cpi is selected. If it not defined then 400 cpi is selected. The default is 800 cpi.

3.3.6.7 *MOUSE_BATTERY_STATUS*

Enabling this feature causes the battery level measurement code to be compiled into the mouse image. The ADC block must be configured properly in the PSoC Designer layout. The mouse then measures the battery level and reports any changes to the bridge. Notification of the battery level is done at the following events: the battery level changes, the mouse transitions from the idle state to the active state, mouse transitions from the disconnected state to the connected state.

3.3.6.8 *MOUSE_TEST_MODE*

This configuration definition is used to selectively compile code for mouse test mode. If this value is defined, then the test mode is compiled into the executable image.

The test mode moves the mouse in a fashion to repeatedly draw the letters 'LP' in a drawing program. Mouse acceleration or advanced motion must be turned off when performing this test. See the **testmode** module for more information on entering this test mode.

3.3.6.9 *MFG_TEST_CODE*

This configuration definition is used to selectively compile in the manufacturing test code. The manufacturing test code in this mouse is compatible with the CY3631 Manufacturing Test Kit offered by Cypress Semiconductor. See the **mfgtest** module for a description of how this test mode is executed. See the CY3631 Manufacturing Test Kit documentation for a description of the test operation.

3.3.6.10 *MFG_TX_MODES*

When the MFG_TEST_CODE is defined, then the definition of this name adds in a carrier and random data TX test option. See the **mfgtest** module for more information on these TX modes.

3.3.6.11 *DEVICE_TYPE*

This definition is used by the **protocol** module when filling out the packet header. The value must be set to three for a mouse.

3.3.6.12 *APP_TX_PACKET_SIZE*

This is the application payload size of the data packet. It must be set to a value large enough to hold the largest mouse data packet. Care must be taken when setting the value, since larger values affect memory usage. This value is used in a union with other packet types that are used for the protocol, so the overall memory usage may be larger than what is specified here.

3.3.6.13 *APP_RX_PACKET_SIZE*

This is currently not used by the mouse, but must be set to at least one for compilation. This value is used in a union with other packet types, so the resulting receive packet will likely be larger than one.

3.3.7 Platform and Architecture Portability

The mouse firmware was designed to be easily ported from one hardware platform to another platform with a simple re-mapping of pins on the enCoRe III LV. The file *pdc9302.h* maintains the pin mapping definitions that are used throughout the code and is included in about every file by using the macro *PLATFORM_H* that is defined in *config.h*.

Porting the code to another microprocessor architecture requires modification or leverage of the existing code for processor specific features, along with pin definitions.

3.3.8 Initialization

Initialization of the enCoRe III LV chip is done by code that is generated in *boot.asm* by the PSoC Designer software. The module *boot.asm* calls *main*() in the **mouse** module once the enCoRe III LV has been configured and initialized.

3.3.9 Wireless Protocol Data Payload

The mouse protocol has been optimized to reduce the ‘on-time’ of the radio, which equates to reduced power consumption. This optimization relies upon the RDK requirement of a three-button mouse. With this requirement, it is possible to combine the z-wheel and the button report into a single byte, allowing five bits of information for the z-wheel and three bits for the buttons.

The **protocol** code module offers the ability to send variable length packets, thereby allowing a reduced number of bytes to be transmitted over the air, in order to extend battery life.

Since mouse usage data demonstrates that X, Y optical sensor data is more frequent than z-wheel or button presses, the following transmission packet formats are implemented in this RDK. The packet formats only show the application payload and do not show the protocol packet format.

3.3.9.1 Packet Format 1

When there is only X, Y delta data, the transmitted packet is two bytes.

Table 3-2. Packet Format 1

Byte 1	Byte 2
X Delta (8 bits)	Y Delta (8 bits)

3.3.9.2 Packet Format 2

When there is either z-wheel data or button data, the transmitted packet is three bytes. In the case where there is no X, Y delta data, but there is z-wheel or button data, the X, Y delta bytes will be set to zero. The z-wheel data is a signed value with bit 4 as the sign bit.

Table 3-3. Packet Format 2

Byte 1	Byte 2	Byte 3
X Delta (8 bits)	Y Delta (8 bits)	Buttons (Bits[7:5]), Z Delta (Bits[4:0])

3.3.9.3 Packet Format 3

When battery voltage level is communicated, the transmitted packet is 1 byte.

Table 3-4. Packet Format 3

Byte 1
Battery Level (1 – 10)

3.3.10 Interrupt Usage and Timing

In the RDK mouse, the following interrupts has been enabled:

- Motion interrupt from the optical sensor
- Button (Left, Middle and Right buttons) interrupt
- Bind button interrupt

The **ISR** module provides an interface to enable/disable individual GPIO interrupts. It also provides the top level GPIO interrupt handler for the application. The other modules handle User Module specific interrupts.

The top level handler *isr_gpio*() calls a routine in the GPIO interrupt User Module. The parameter passed is a pointer to another C function that has been declared as an interrupt handler. This is done in order to preserve the registers used in the handler. The *gpio_isr_redirector*() function manipulates the stack, preparing it for the interrupt handler being called. The rule is that the module routine being called may not call any other functions and it must be declared with the #pragma interrupt_handler. For example, the interrupt handler in the **optical** module that handles detection of optical motion does not call any other C functions. Instead, it uses a macro provided by the mouse module to post an event flag. See the **optical/mouse** modules. This method prevents the compiler from pushing all of its virtual registers onto the stack when a function is called at interrupt context.

The interrupt latency includes two portions. The first portion is the time between the assertion of an enabled interrupt and the start of its ISR, which can be calculated using the following equation:

Latency1 = Time for current instruction to finish +
Time for M8C to change program counter to interrupt address +
Time for LJMP instruction in interrupt table to execute.

For example, if the 5-cycle JMP instruction is executing when an interrupt becomes active, the total number of CPU clock cycles before the ISR begins would be as follows:

(1 to 5 cycles for JMP to finish) +
(13 cycles for interrupt routine) +
(7 cycles for LJMP) = 21 to 25 cycles.

In the example above, at 12 MHz, 25 clock cycles take 2.083 µs.

The second portion is the time between the start of the ISR and the post of the event flag. For example, the motion interrupt takes 308 CPU clock cycles for this portion. Therefore, the Latency2 equals to 25.667 µs for the 12 MHz CPU.

Consequently, the total latency for a motion interrupt is:

Latency1 + Latency2 = 27.750 µs

3.3.11 Code Performance Analysis

A mouse motion report is used to analyze the code performance. A typical mouse motion report contains the following steps:

- Optical sensor responds to a mouse motion. With the mouse the sensor in the rest 1 state, it takes 16.5 ms for the sensor to responds to this sensor motion.
- The sensor interrupts the MCU by lowering its motion pin. The prior section has calculated that it takes 27.750 µs for MCU to respond to this Interrupt.
- In the function timer_wait_event (), the MCU exits the sleep state and spends 53 µs to finish the wheel poll.

- Firmware delays MOUSE_REPORT_IN_MS, which is 10 ms for the default. This delay is to prevent excessive radio retries from the mouse.
- Firmware calls function mouse_do_report() to read the Delta_X and Delta_Y value and send the packet to the bridge. This step takes 1.98 ms, which includes 1.66 ms radio transmission time.

As a result, if a mouse is in the rest 1 state, it takes 28.6 ms for the mouse to report a motion to a bridge.

3.4 Development Environment

See the *CY4672 Getting Started Guide* for a list of tools required to build and debug the mouse application.

3.4.1 Tips and Tricks

3.4.1.1 M8C Sleep

When using the ICE-Cube, define the macro PSOC_ICE so that busy waits are used instead of the sleep instruction. Using the sleep instruction with the ICE-Cube generates errors due to synchronization issues between the OCD part and the emulator.

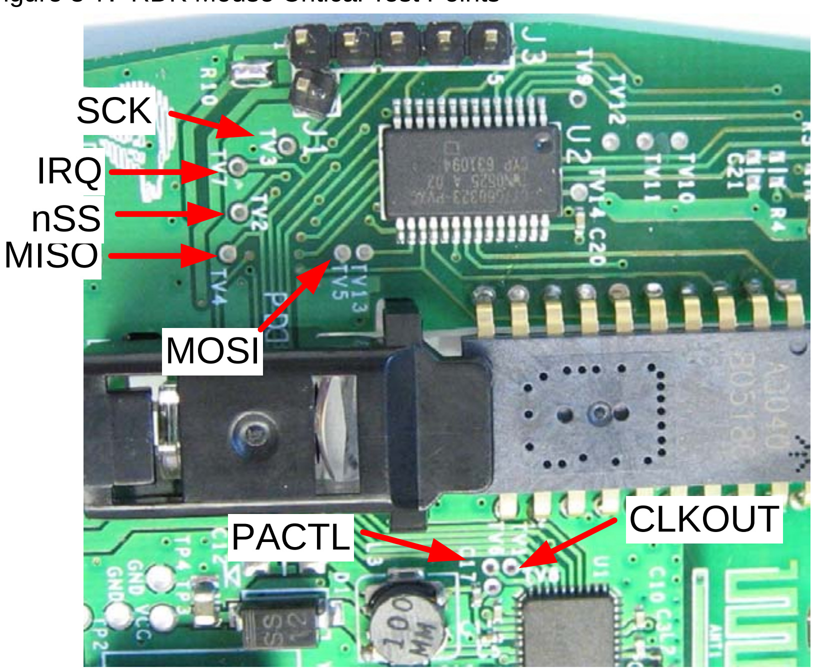
3.4.1.2 Watchdog Timer

The watchdog timer is enabled for the RDK operation, but may be disabled for debug purposes.

3.4.2 Critical Test Points

The following figure shows the critical test points for RDK mouse.

Figure 3-7. RDK Mouse Critical Test Points



4. Keyboard

4.1 Introduction

This section covers the design goals, architecture, firmware source code modules, and configuration options for the WirelessUSB LP keyboard. It does not cover the details of the radio subsystem or the configuration options that go with it.

4.1.1 Design Features

There are several design goals that drove the requirements for the firmware development for the keyboard. Some of these are architecture related, while others are feature related.

The CY4636 Reference Design Kit uses a enCoRe II LV controller for the RDK keyboard. Contact your local sales representative for more information on the enCoRe II LV controller.

The architecture was designed to be modular for extendibility and maintainability. It was also designed so that it could easily be ported from one hardware platform to another assuming the use of a enCoRe II LV microprocessor. While porting to another microprocessor requires more work, the hardware design was done to minimize usage of advanced enCoRe II LV features to expedite this effort.

Design efforts have been made to reduce the ON time of the microprocessor and radio to conserve battery life. This includes protocol optimizations along with using sleep features of the radio and enCoRe II LV microprocessor.

4.2 Hardware Overview

Several pictures of the RDK keyboard assembly are presented in this section; also the schematics, a keyboard matrix table, and hardware considerations are discussed in this section.

4.2.1 RDK Keyboard Assembly

Figure 4-1. Keyboard Plastic

Figure 4-1 shows the RDK keyboard plastic.

Figure 4-2. Exploded Keyboard

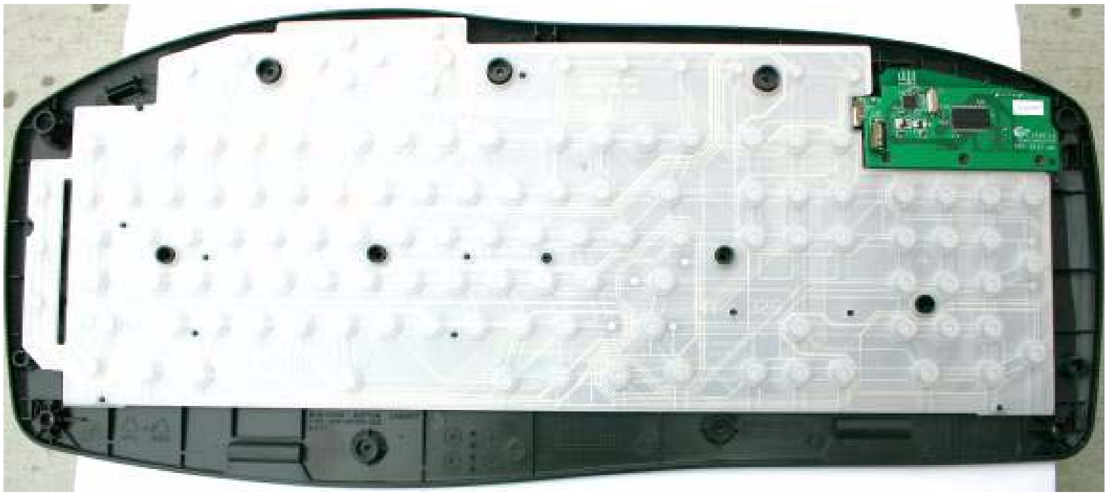


Figure 4-2 shows the keyboard with the top removed. The radio/ enCoRe II LV board (PDC-9265) is shown in the upper right hand corner.

Figure 4-3. Radio and PSoC Board (PDC-9265)

Figure 4-3 shows the main controller board with the enCoRe II LV and WirelessUSB LP Radio. All of the components are on the top side of the board with the exception of the Bind button.

Figure 4-4. Keyboard Battery Compartment

Figure 4-4 shows the integrated battery compartment located on the bottom side of the keyboard. The battery compartment cover is also shown.

Figure 4-5. Bind Button

Figure 4-5 shows the Bind button.

4.2.2 Schematic

All schematics for the LP RDK keyboard are located in the following directory: `<installation directory>\Hardware\keyboard`. The schematic is in Adobe Acrobat format with the letters 'Sch' in the file name.

4.2.3 Keyboard Matrix

The RDK keyboard matrix has 18 columns and 8 rows. Key presses generate a GPIO interrupt when a column is connected (shorted) to a row. The keyboard then scans the matrix to determine which keys have been pressed.

The RDK keyboard matrix with the USB scan codes are shown in Table 4-1.

Table 4-1. RDK Keyboard Matrix

	Row 0	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
Column 0	0x09	0x0A	0x19	0x05	0x17	0x15	0x21	0x22
Column 1	0x0D	0x0B	0x10	0x11	0x1C	0x18	0x24	0x23
Column 2	0x0E	0x3F	0x36	NA	0x30	0x0C	0x25	0x2E
Column 3	0x0F	NA	0x37	NA	0x40	0x12	0x26	0x41
Column 4	0x33	0x34	NA	0x38	0x2F	0x13	0x27	0x2D
Column 5	0x31	0x3E	0x28	0x2C	0x2A	NA	0x43	0x42
Column 6	0x5A	0x62	0x54	0x4F	0x5D	0x60	0x45	0x49
Column 7	0x59	NA	0x53	0x51	0x5C	0x5F	0x44	0x4C
Column 8	0x5B	0x63	0x55	0x56	0x5E	0x61	0x4E	0x4B
Column 9	0x07	0x3D	0x06	NA	0x3C	0x08	0x20	0x3B
Column 10	0x16	NA	0x1B	NA	0x39	0x1A	0x1F	0x3A
Column 11	0x04	0x29	0x1D	NA	0x2B	0x14	0x1E	0x35
Column 12	0x58	0x52	0x48	0x50	NA	0x57	0x4D	0x4A
Column 13	NA	0x04	NA	0x40	0x0192	0x47	0x46	0x0223
Column 14	0x02	0x00CD	0x20	NA	0x02	NA	0x0221	0x018A
Column 15	NA	NA	0x10	NA	0x00E9	NA	NA	0x01
Column 16	0x7D	0x00E2	0x80	0x7C	0x00B7	0x00EA	0x022A	NA
Column 17	0x08	0x0225	NA	0x7B	0x0224	0x65	0x00B6	0x00B5

Notes:

- Yellow indicates Multimedia Key (16-bit value)
- Red indicates Power Key
- Blue indicates Modifier Key
- No color indicates a Standard 101 Key

4.2.4 Hardware Considerations

The keyboard design uses the BAT400D-7-F schottky diode (D1) and CDH53100LC inductor (L3) for its boost circuitry. These low cost components are used to reduce the over all system cost at the expense of lower boost efficiency. Preliminary characterization data shows a range of 68-81% efficiency for the 1.8-2.7V VBAT voltage range at different temperatures (-10C to 80C). Higher efficiency components such as the ones in the mouse design may be used at the expense of component costs and board size (these low cost components are smaller in size compared to the ones used in the mouse design.)

4.3 Firmware Architecture

There are two architectural views of the keyboard. The first is a microcontroller configuration view of User Modules. This architecture and configuration is best viewed in the PSoC Designer application when the project is loaded. The second view is a logical organization of the source code modules that make up the keyboard application code and other support modules.

The next few sections describe both architectures with emphasis on top level organization and overall module operation. Obtain more detailed description of variables and functions by referencing the source code.

4.3.1 ROM/RAM Usage

The following table shows the ROM/RAM usage. The top part exhibits the total ROM/RAM usage for basic functions, which disables all the build options below. The bottom part exhibits the ROM/RAM usage for individual build options.

Table 4-2. ROM/RAM Usage

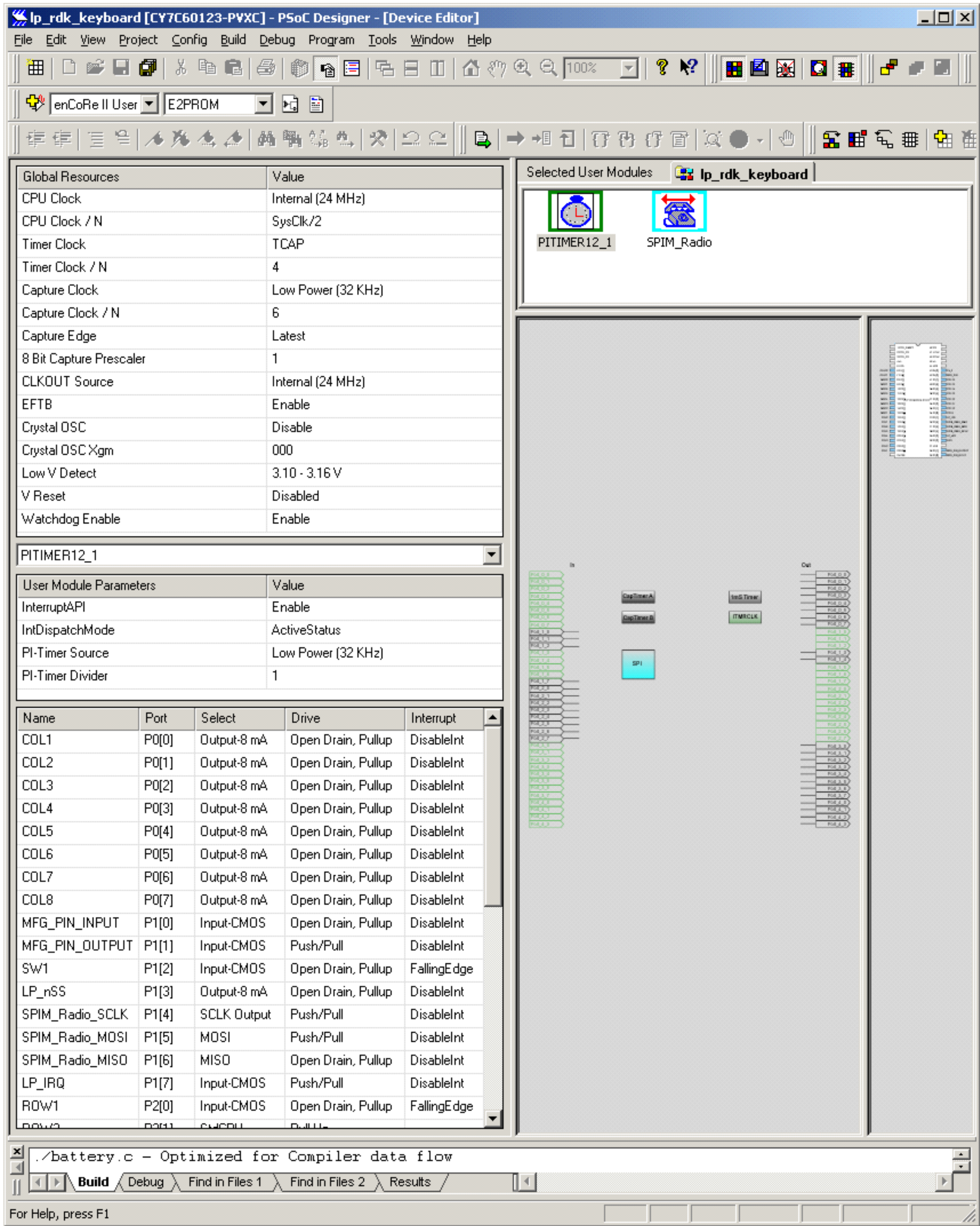
	Total ROM (Bytes)	**Total RAM (Bytes)**
Basic Functions	5550	119
Build Option	**ROM Usage (Bytes)**	**RAM Usage (Bytes)**
KEYBOARD_MULTIMEDIA_SUPPORT	755	1
KEYBOARD_TEST_MODES	338	1
KEYBOARD_BATTERY_VOLTAGE_SUPPORT	166	1
KEYBOARD_FAST_SCAN	124	0
ENCRYPT_DATA*	942	1
MOUSE_EMULATION_MODE	610	1
MFG_TEST_CODE	457	0
MFG_TX_MODES	687	3

*The ENCRYPT_DATA option requires 64 bytes extra ROM space to store the non-volatile session key.

4.3.2 enCoRe II LV Device Configuration

The enCoRe II LV is configured using the Device Editor in PSoC Designer. The Device Editor allows the Global Resources for the part and user module parameters to be configured. The keyboard uses two separate user modules. The first module is an SPI master for communicating with the keyboard and the radio. The second module is a programmable interval timer. The following is a screen shot of the Device Editor showing the User Module mapping. Further description of resources and User Modules follow the diagram.

Figure 4-6. Microcontroller Device Architecture



4.3.2.1 *Global Configuration*

The following is a description of the **Global Resources** that are configured for the CY7C60123-PVXC enCoRe II LV microcontroller. Care must be taken when modifying these values as they affect the User Modules discussed below.

4.3.2.1.1 CPU Clock

This parameter is set to Internal (24 MHz). In order to run the CPU at 12 MHz, CPU Clock/N must to be set to '2'. This operating frequency provides for faster code execution so that when events are detected the microcontroller can be put back into the sleep state quicker for improved power savings.

4.3.2.1.2 CPU Clock / N

This parameter is set to 2 to provide a 12 MHz clock.

4.3.2.1.3 Timer Clock

This parameter is set to TCAP.

4.3.2.1.4 Timer Clock /N

This parameter is set to 4.

4.3.2.1.5 Capture Clock

This parameter is set to Low Power (32 kHz).

4.3.2.1.6 Capture Clock /N

This parameter is set to 6.

4.3.2.1.7 Capture Edge

This parameter is set to Latest.

4.3.2.1.8 8 Bit Capture Prescaler

This parameter is set to 1.

4.3.2.1.9 CLKOUT Source

This parameter is set to Internal (24 MHz).

4.3.2.1.10 EFTB

This parameter is set to Enable.

4.3.2.1.11 Crystal OSC

This parameter is set to Disable.

4.3.2.1.12 Crystal OSC Xgm

This parameter is set to 000.

4.3.2.1.13 Low V Detect

This parameter is set to 3.10-3.16 V.

4.3.2.1.14 V Reset

This parameter is set to Disabled.

4.3.2.1.15 Watchdog Enable

This parameter should be set to Enable, but may be set to Disable for debug purposes.

4.3.2.2 SPI Master User Module

The SPI Master User Module is used to communicate with the LP radio. The LP radio supports leading edge data latching, non-inverted clock, and MSB first transmission as defaults. A clock divider of 6 is chosen which generates an SPI clock of 2 MHz. The interrupt API to this module is not used.

4.3.2.3 *Programmable Interval Timer User Module*

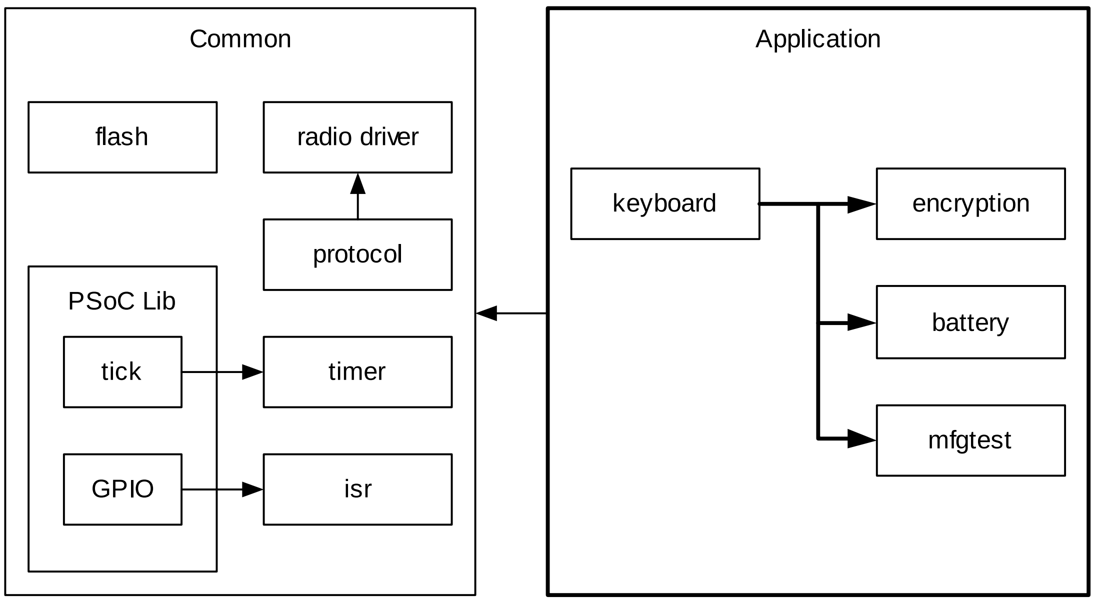
This module is used for timing.

4.3.2.4 *Flash Security*

The keyboard project within PSoC Designer has a file called *FlashSecurity.txt*. This file specifies access rules to blocks of Flash ROM. See the documentation at the top of the file for definitions. This file is shipped with a single change from its default configuration. The blocks starting at address 1F80 hex are changed from W: Full (Write protected) to U: Unprotected. These locations of Flash are dedicated to save non-volatile configuration values for the **protocol** code module and non-volatile session key for the **encrypt** code module.

4.3.3 Model

Figure 4-7. Firmware Architecture Model



The keyboard firmware is partitioned into two logical groups. The Common group is a collection of code modules that provide the underlying support for the application. This group provides services such as, radio protocol, radio driver, timing, flash access, and interrupts.

The Application group implements the core functionality and features of RDK wireless keyboard. This includes power management, encryption, packet formatting and reporting, various test modes, and battery level sensing. The code modules for each of these groups are described below in further detail.

All of the following module descriptions have corresponding *<module name>.c* or *<module name>.asm* and *<module name>.h* source code files. The module API and definitions are exported in the header file while the module implementation and local definitions are contained in the C/assembly file.

4.3.4 Common Code

The modules in the common code group are a combination of two sources. The first is PSoC Designer generated files in the *lib* directory that have been modified to support the application. The second group is modules that are generally used by the application.

4.3.4.1 Generated Library Code

There are currently no files, generated by PSoC Designer, that are modified for the use of the application.

4.3.4.2 Radio Driver

The radio driver module is a low level module providing basic radio communication and configuration. Its general application is such that it is likely not to be changed by the firmware developer. It provides an interface for reading/writing radio registers, setting PN codes and initialization of the radio and transmitting or receiving packets. See the Radio Driver documentation for details.

4.3.4.3 Protocol Module

The protocol module defines and implements the layer used to deliver packets from the device to the bridge. It manages the binding of devices to a bridge as well as the connection and interference immunity by channel hopping. This module has a dependency on the **radio driver** for sending and receiving formatted packets and the **flash** module for storing the manufacturing ID of the bridge the device is bound to.

4.3.4.4 Flash Module

The flash module is a smaller version of the E2PROM module provided in PSoC Designer. It is limited in functionality and only implements the read/write routines required by the device. Modify the *flashsecurity.txt* file so that the block being modified by this module is given read/write privilege, i.e. unprotected. Currently the two very top most blocks in flash are used by this module. One block is used for storing the encryption key if encryption is enabled. The other block is used to store the bind parameters.

4.3.4.5 ISR Module

This module provides an interface to initialize the interrupt.

4.3.4.6 Timer Module

The timer module provides a one-millisecond tick for the system. The tick resolution can be changed, but is set for one millisecond for the keyboard. This module requires the use of a 12-bit Programmable Interval Timer user module of the enCoRe II LV. The delay function used for millisecond timing provides at least the delay requested with no more than one additional millisecond of delay. The millisecond delay function will sleep the PSoC for the duration of the requested delay. The microprocessor wakes just long enough to update the tick every millisecond and check if the delay has been met and then returns to sleep state if it has not. See the documentation in the module for requirements on configuring the enCoRe II LV block.

4.3.5 Application Code

The group of modules that make up the application code is responsible for implementing the keyboard functionality and behavior. Following is a high level description of each module responsibility and associated algorithms.

4.3.5.1 *Keyboard Module*

The keyboard module is the controlling code for the application. It has many responsibilities in implementing various features and functions offered by the keyboard.

A few types are defined to support the operation of the keyboard. One of these is the packet format used when sending data to the bridge. This type is defined as TX_PACKET and is structured to support the different data packet formats as explained in the section Wireless Protocol Data Payload on page 55.

The function *main*() is the entry point for the keyboard application. This function is called from the *boot.asm* file. The keyboard first initializes all of the application modules and then initializes the **protocol** module. There is an order dependency for some of these, so care must be taken in modifying the *keyboard_init*() function. For example, other modules depend upon the timer facility running in order to perform initialization. Once each module has been initialized, then the application checks for entry to the manufacturing test mode. If the manufacturing test mode is not indicated, then normal keyboard operation begins.

There are two states for the keyboard operation: the idle state and the active state. The keyboard initially enters idle state; when there is any keystroke, it enters the active state.

In active state the keyboard is scanned for both the keys and the Bind button. The keystrokes are collected, formatted and reported to the bridge. After that, the keyboard goes into the idle state.

In idle state the MCU and radio go to sleep to save power, and the keyboard application remains waiting for input from the keys or Bind button. The timer is turned off to conserve power. This state is maintained indefinitely until a keystroke or a button press occurs.

The battery level is reported by the keyboard application when it detects a keystroke after it has been in an idle state for 8 seconds.

In the active state the keyboard attempts to deliver a packet for the amount of time designated in KEYBOARD_TX_TIMEOUT. The keyboard application is also responsible for detecting the Bind button press, and then calling the bind function in the **protocol** module.

The keyboard application sends keyboard reports as frequently as events arrive, but not any faster than the time defined in the macro KEY_DOWN_DELAY_SAMPLE_PERIOD. Carefully set this time so that the report rate does not exceed that which the USB bus is capable of handling. Keep in mind that the report rate varies slightly due to drift of the internal oscillator used to keep track of time.

4.3.5.2 *Mfgtest Module*

The RDK provides a compile-time option of adding a manufacturing test mode to the keyboard. The manufacturing test code in this keyboard is compatible with the CY3631 Manufacturing Test Kit offered by Cypress Semiconductor.

If MFG_TEST_CODE is defined and ENTER_BY_PIN is not defined, holding down the system sleep key and the Bind button while inserting the batteries into the keyboard will enter the manufacturing test mode.

If MFG_TEST_CODE and ENTER_BY_PIN are both defined, connecting pins 4 and 5 on the ISP header with a shunt and then inserting the batteries into the keyboard will enter the manufacturing test mode.

The only way to exit this mode is to cycle power.

4.3.5.3 *Battery Module*

The battery monitor circuit is implemented using the Low Voltage Interrupt (LVI) on the LP radio. Following is an explanation of the process to measure the battery voltage.

The process first sets the LVI threshold to 1.8V and then checks for an LVI interrupt. If the interrupt does not occur then it repeatedly sets the LVI TH and PMU OUTV with the following combination and checks the status.

Table 4-3. LVI TH and PMU OUTV Combinations

LVI TH	PMU OUTV	VOLTAGE IF INTERRUPT OCCURS
1.8V	2.7V	< 1.8V
2.0V	2.7V	< 2.0V
2.2V	2.7V	< 2.2V
PMU OUTV	2.7V	< 2.7V

It then returns a battery level between 1 and 10: 1 being below 1.8V and 10 being above 2.7 volts.

4.3.5.4 Test Module

This RDK keyboard provides a compile-time option of adding test modes to the keyboard; see section KEYBOARD_TEST_MODES on page 53 for enabling this option. The test mode module is implemented in a way that it can be easily extended to add other test modes. Currently there are only two test modes supported in the module. When this option is not enabled then all test mode code is removed from the compilation.

The first test mode is initiated by holding down the left ctrl, left alt, right alt, right ctrl, and F1 keys at the same time. If PANGRAM_TEST_MODE is defined, the test sends the key up/down scan codes for the test pangram: *"a quick brown fox jumps over the lazy dog.<carriage return>"* . Otherwise the up/down scan codes are repeatedly sent for the test sequence 'wirelessusb' followed by the same number of backspaces. The test repeats the appropriate sequence until the escape key is pressed. Once the test has finished execution, the keyboard returns to normal operation.

The repeating 'x' test selection is initiated by holding down the left ctrl, left alt, right alt, right ctrl, and F3 keys at the same time. The test continuously sends the 'x' key up/down scan codes. The test continues until the escape key is pressed. Once the test has finished execution, the keyboard returns to normal operation.

4.3.5.5 Encrypt Module

This module may be conditionally compiled in to provide encryption/decryption support. Encrypted data transfers are typically used between RDK keyboard devices and the RDK bridge. Contact Cypress Applications support for the encryption source code.

4.3.6 Configuration Options

All configuration options for the application can be found in the *config.h* file. Each item is explained below and can be changed to values that meet the developer's requirements.

4.3.6.1 KEYBOARD_KEEP_ALIVE_TIMEOUT

When a key is held down, this configuration value sets the period at which the firmware generates a KEEP_ALIVE packet since the last keyboard report. The default is 65 milliseconds.

4.3.6.2 KEY_DOWN_DELAY_SAMPLE_PERIOD

This configuration value sets the period at which the firmware polls the hardware for keyboard events to transmit over the radio. This poll period is only active when the keyboard has not entered sleep because keys are currently being pressed. The default value is 10 milliseconds.

4.3.6.3 KEYBOARD_DEBOUNCE_COUNT

The button debounce logic detects changes in the button state and immediately indicates a change causing a report to be sent to the radio. The debounce logic then blocks out any further button state changes for the specified debounce time. This operation is somewhat different from the usual method of waiting for a button to stabilize during a debounce interval, and then reporting the change in button state. It is implemented this way to improve button-reporting latency.

This configuration value sets the debounce time for buttons that are pressed. It is measured in units of the poll rate. For example, if KEYBOARD_DEBOUNCE_COUNT is defined as 2 and KEY_DOWN_DELAY_SAMPLE_PERIOD is defined as 10, then the button debounce time will be 20 milliseconds. The default setting is ‘2’.

4.3.6.4 KEYBOARD_MULTIMEDIA_SUPPORT

This configuration definition is used to selectively compile support for multimedia (hot) keys. If this value is defined, then multimedia key support is compiled into the executable image. If it is not defined, then the multimedia support code is omitted.

4.3.6.5 KEYBOARD_TEST_MODES

This configuration definition is used to selectively compile code for keyboard test modes. If this value is defined, then test modes are compiled into the executable image. If it is not defined, then the test mode code is omitted. The test modes are described in section Test Module on page 52.

4.3.6.6 KEYBOARD_TEST_MODE_PERIOD

This configuration value sets the period that the keyboard generates on test key presses. A key press consists of a scan code as the down key and a NULL as the up key. The default value is 10 ms.

4.3.6.7 PANGRAM_TEST_MODE

This configuration definition is used to selectively compile in the pangram test mode. A pangram is a sentence that contains all of the letters of the alphabet at least once.

4.3.6.8 KEYBOARD_BATTERY_VOLTAGE_SUPPORT

This configuration definition is used to selectively compile support for battery voltage level reporting. If this value is defined, then battery voltage level reporting is compiled into the executable image. If it is not defined, then the battery voltage level reporting code is omitted.

4.3.6.9 LP_RDK_KEYBOARD_MATRIX

This configuration definition is used to selectively compile in the keyboard matrix for the RDK keyboard hardware.

4.3.6.10 KEYBOARD_FAST_SCAN

This configuration definition is used to selectively compile in the Cypress Semiconductor fast scan algorithm. Fast Scan is used to minimize the time it takes for the CPU to scan the key matrix which in turn reduces the current consumption.

4.3.6.11 KEYBOARD_TX_TIMEOUT

This configuration value sets the maximum time that the keyboard tries to send a report to the bridge. The default value is 5000 ms.

4.3.6.12 TIMER_CAL

This configuration definition is used to selectively compile in the one-millisecond timer calibration routine. The routine is called on power on and during protocol reconnect.

4.3.6.13 ENCRYPT_DATA

This configuration definition is used to selectively compile in data encryption for the keyboard. Contact Cypress Applications support for the encryption source code.

4.3.6.14 MFG_TEST_CODE

This configuration definition is used to selectively compile in the manufacturing test code. The manufacturing test code in this keyboard is compatible with the CY3631 Manufacturing Test Kit offered by Cypress Semiconductor. See the **mfgtest** module for a description of how this test mode is executed. See the CY3631 Manufacturing Test Kit documentation for a description of the test operation.

4.3.6.15 MFG_ENTER_BY_PIN

This configuration definition is used to select whether the manufacturing test code is executed by connecting pin 4 and 5 on the ISP (programming) header. When this value is not defined, then the manufacturing test code may be executed by holding the system sleep key and the Bind button when the batteries are inserted into the keyboard.

4.3.6.16 MFG_TX_MODES

When the MFG_TEST_CODE is defined, the definition of this name adds in a carrier and random data TX test option. See the **mfgtest** module for more information on these TX modes.

4.3.6.17 MOUSE_EMULATION_MODE

This configuration definition is used to selectively compile in the mouse Emulation Mode. The Scroll Lock key is used to toggle this mode on/off. Once in this mode, the arrow keys are used to move the mouse. The Delete key is the left mouse button, the End key is the right mouse button, and Page Up and Page Down emulate the scroll wheel.

4.3.6.18 KEYBOARD_POWER_ON_BIND

This configuration definition is used to selectively compile in the option to enter bind mode on power-up when the device has not been previously bound to a bridge.

4.3.6.19 PLATFORM_H

This configuration value identifies the header file that has the platform configuration information. The default value is *pdc9265.h*, which is identifier for the keyboard board that is shipped with the RDK. It is anticipated that this macro will change when the code is ported to another platform.

4.3.7 Platform and Architecture Portability

The keyboard firmware was designed to be easily ported from one hardware platform to another platform with a simple re-mapping of pins on the enCoRe II LV. The file *pdc9265.h* maintains the pin mapping definitions that are used throughout the code and is included in about every file by using the macro *PLATFORM_H* that is defined in *config.h*.

The keyboard scan matrix is defined in *kdefs.h* and may need to be changed for different keyboards.

Porting the code to another microprocessor architecture requires modification or leverage of the existing code for processor specific features, along with pin definitions.

4.3.8 Initialization

Initialization of the enCoRe II LV chip is done by code that is generated in *boot.asm* by the PSoC designer software. The module *boot.asm* calls main once the enCoRe II LV has been configured and initialized.

Main initializes the components of the keyboard along with timer, isr and radio modules. The main routine then goes into an infinite loop monitoring keyboard activity and sleeping between keystrokes.

4.3.9 Wireless Protocol Data Payload

The keyboard protocol has been optimized to reduce the ON time of the radio and power consumption.

The radio driver offers the ability to send variable length packets, allowing the opportunity to minimize the number of bytes transmitted over the air, in order to extend battery life.

The following transmission packet formats are implemented in this RDK. The report formats show the application payload and the radio protocol overhead with example packet headers.

4.3.9.1 Keyboard Application Report Formats

The first byte of the data packet payload, byte 2 of the radio packet, is used as a keyboard application report header. There are five possible keyboard application reports. The reports are:

- Standard 101 Keys Report
- Multimedia Keys Report
- Power Keys Report
- Keep Alive Report
- Battery Voltage Level Report

The first application report byte is Scan Code 1 if the byte is less than 0xFC. Otherwise, the first application report byte is the Application Report Header (Multimedia, Power, Battery, or Keep Alive). This also assumes that multimedia and power keys do not use modifier keys and that 0xFF, 0xFE, 0xFD and 0xFC are not valid Standard 101 key scan codes.

Trailing zeros in the reports are also removed to further minimize the number of bytes sent by the radio.

The LP radio sends the reports with the format shown in Table 4-4.

Table 4-4. LP Generic Report

Byte	1					2	..	..	N
Bits:	7:4	3	2	1	0	7:0	7:0	7:0	7:0
Field:	4	0	Toggle	ID	0	Application Report Header			Byte N

4.3.9.1.1 Standard 101 Keys Report

If the Application Report Header byte is less than 0xFC then this indicates that this report is a Standard 101 Keys Report and the first byte is the actual scan code rather than the Report Header. This is done to optimize the packet size based on the fact that the most common report will have only one non-zero scan code without a modifier. The full Standard 101 Keys report format is shown in Table 4-5.

Table 4-5. Standard 101 Keys Report Format

Byte	Name
2	Scan Code 1 (< 0xFC)
3	Modifier Keys
4	Scan Code 2
5	Scan Code 3
6	Scan Code 4
7	Scan Code 5
8	Scan Code 6

Example: The following reports would be sent if a user presses an ‘a’ on the keyboard. The down key packet sent from the keyboard to the bridge is shown in Table 4-6.

Table 4-6. Example ‘a’ down key Standard 101 Keys Report

Byte 2
Scan Code 1
0x04

The bridge would add the trailing zeros, insert the reserved byte, rearrange the modifier and scan code 1 bytes and remove the packet header to produce the USB report shown in Table 4-7.

Table 4-7. Example USB report for the ‘a’ down key

Modifier keys	Reserved	Scan Code 1	Scan Code 2	Scan Code 3	Scan Code 4	Scan Code 5	Scan Code 6
0x00	0x00	0x04	0x00	0x00	0x00	0x00	0x00

The up key packet sent from the keyboard to the bridge (all data bytes are zero) is shown in Table 4-8.

Table 4-8. Example up key Standard 101 Keys Report

Byte 2
Scan Code 1
0x00

The bridge would add the trailing zeros, insert the reserved byte, and remove the packet header to produce the USB report shown in Table 4-9.

Table 4-9. Example USB report for a Standard 101 Key Null Packet Report

Modifier keys	Reserved	Scan Code 1	Scan Code 2	Scan Code 3	Scan Code 4	Scan Code 5	Scan Code 6
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

4.3.9.1.2 Multimedia Keys (Hot keys) Report

An Application Report Header of 0xFF indicates that this report is a Multimedia Keys Report. The Multimedia Keys report format is shown in Table 4-10.

Table 4-10. Multimedia Keys Report Format

Byte	Name
2	Application Report Header 0xFF
3	Hot Key Scan Code (upper 8 bits)
4	Hot Key Scan Code (lower 8 bits)

Example: The following reports would be sent if a user presses 'Volume Increase' (Hot Key 8) key on the keyboard.

The 'Volume Increase' down key packet sent from the keyboard to the bridge is shown in Table 4-11.

Table 4-11. Example 'Volume Increase' down key Multimedia Keys Report

Application Report		
Application Report Header	Hot Key Scan Code (upper 8 bits)	Hot Key Scan Code (lower 8 bits)
0xFF	0x00	0xE9

The up key packet sent from the keyboard to the bridge is shown in Table 4-12.

Table 4-12. Example up key Multimedia Keys Report

Application Report
Application Report Header
0xFF

4.3.9.1.3 Power Keys (Suspend/Sleep) Report

An Application Report Header of 0xFE indicates that this report is a Power Keys Report. The Power Keys report format is shown in Table 4-13.

Table 4-13. Power Keys Report Format

Byte	Name
2	Application Report Header (0xFE)
3	Power Key Scan Code

Example: The following reports would be sent if a user presses the Suspend/Sleep (Power Key 0) key on the keyboard.

The Suspend/Sleep down key packet sent from the keyboard to the bridge is shown in Table 4-14.

Table 4-14. Example Suspend/Sleep Down Key Power Keys Report

Application Report	
Application Report Header	Power Key Scan
0xFE	0x02

The up key packet sent from the keyboard to the bridge is shown in Example Up Key Power Keys Report.

Table 4-15. Example Up Key Power Keys Report

Application Report
Application Report Header
0xFE

4.3.9.1.4 Keep Alive Report

An Application Report Header of 0xFC indicates that this report is a Keep Alive Report.

Example of a Keep Alive reports sent from the keyboard to the bridge is shown in Table 4-16.

Table 4-16. Example Keep Alive Report (Null Packet Support disabled)

Application Report
Application Report Header
0xFC

If the bridge does not receive a Keep Alive packet or an up key within a specified interval (DOWNKEY_TIME_OUT) while a down key is present, the bridge generates an up key to the computer.

4.3.9.1.5 Battery Voltage Level Report

An Application Report Header of 0xFD indicates that this report is a Battery Voltage Level Report. The Battery Voltage Level report format is shown in Table 4-17.

Table 4-17. Battery Voltage Level Report Format

Byte	Name
2	Application Report Header 0xFD
3	Battery Voltage Level

The Battery Voltage Level ranges from 1 (low) to 10 (full).

The Battery Voltage Level Report is sent after a keystroke that occurs whenever the keyboard has been in idle for more than 8 seconds.

Example of a Battery Voltage Level Report with fully charged batteries is shown in Table 4-18.

Table 4-18. Example ‘full’ Battery Voltage Level Report

Application Report	
Application Report Header	Battery Voltage Level
0xFD	0x0A

Example of a Battery Voltage Level Report with low batteries is shown in Table 4-19.

Table 4-19. Example ‘low’ Battery Voltage Level Report

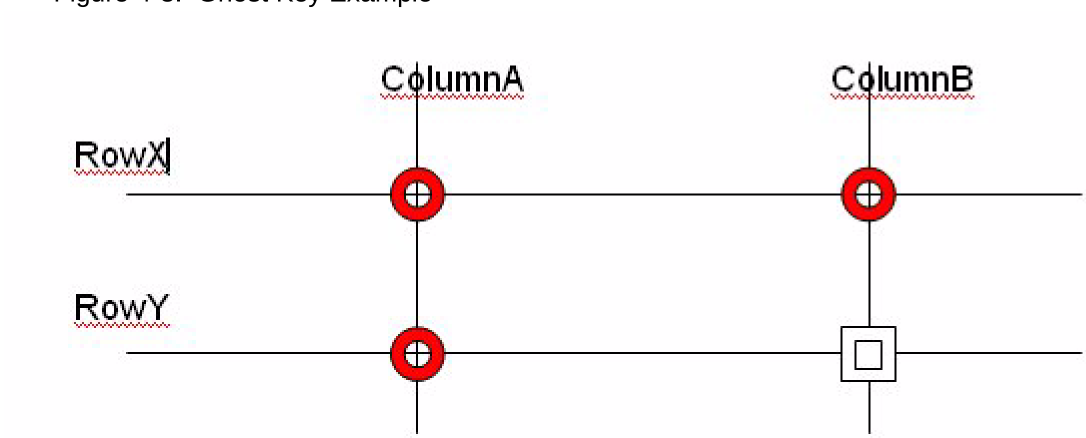
Application Report	
Application Report Header	Battery Voltage Level
0xFD	0x01

4.3.10 Ghost Key Detection

Ghost keys are possible on the RDK keyboard because it does not use diodes with the keyboard switches. Ghost keys are caused when three keys are pressed at the same time and two of the keys are on the same column and two of the keys are on the same row. When scanning the keyboard, it appears that four keys have been pressed and it is impossible to tell which three of the four keys are actually valid. The keyboard code detects this condition and does not send a report until one of the three keys is released.

For example, assume the keys (RowX, ColumnA), (RowX, ColumnB), and (RowY, ColumnA) have been pressed as shown in Figure 4-8. It appears that the key (RowY, ColumnB) has been pressed as well when it has not since the other keys electrically connect RowY to ColumnB.

Figure 4-8. Ghost Key Example



4.3.11 Interrupt Usage and Timing

In the RDK keyboard, the following interrupts have been enabled:

- Row Port interrupt
- Bind button interrupt

When either of the above interrupts occurs, its ISR sets the flag.

The interrupt latency includes two portions. The first portion is the time between the assertion of an enabled interrupt and the start of its ISR, which can be calculated using the following equation:

Latency1 = Time for current instruction to finish +
Time for M8C to change program counter to interrupt address +
Time for LJMP instruction in interrupt table to execute.

For example, if the 5-cycle JMP instruction is executing when an interrupt becomes active, the total number of CPU clock cycles before the ISR begins would be as follows:

(1 to 5 cycles for JMP to finish) +

(13 cycles for interrupt routine) +

(7 cycles for LJMP) = 21 to 25 cycles.

In the example above, at 12 MHz, 25 clock cycles take 2.083 µs.

The second portion is the time between the start of the ISR and the set of the flag. For example, the row port interrupt (caused by pressing any key) takes 19 CPU clock cycles for this portion. Therefore, the Latency2 equals to 1.583 µs for the 12 MHz CPU.

Consequently, the total latency for a button interrupt is

Latency1 + Latency2 = 3.667 µs

4.3.12 Code Performance Analysis

A key press report is used to analyze the code performance. A typical key press report contains the following steps:

- A key press interrupts the MCU. The prior section has calculated that it takes 3.667 µs for MCU to responds to this Interrupt.
- MCU exits the sleep state, scans the Bind button and turns on the timer. It takes 40.8 µs.
- MCU calls function scan_keyboard() to detect which key is pressed. This function consumes 1.15 ms.
- MCU calls function generate_standard_report () to format the report and send the report to the bridge. This step takes 2.01 ms, which includes 1.66 ms radio transmission time.

As a result, it takes 3.20 ms for the keyboard to report a key press to the bridge.

4.4 Modifying the Keyboard Matrix or Adding New Keys

The current keyboard matrix with the USB scan codes are shown in Table 4-1 on page 45. Customers may modify the keyboard matrix or they may add new keys to their keyboard. The following sections explain the procedure.

4.4.1 Modifying the Keyboard Matrix

In the file *kdefs.h*, a table called default_keyboard_scan_table matches the keyboard matrix shown in Table 4-1 on page 45. By modifying this table, the keyboard matrix is automatically modified.

4.4.2 Adding New Keys

Example: The customer wants to add a multimedia key called **My Computer**, which is located at Column 15 and Row 6 and has a scan code of 0x0194. The following steps must be performed:

1. Go to file *kdefs.h*, and search for default_keyboard_scan_table. In the Col 15 (0xF) section, modify line 7 from {NO_DEVICE, NOKEY} to {DEVICE_2, 0x000E}. The 0x000E is the index into the device 2 table.
2. Go to the table called device_2_keyboard_scan_table within the same file and add the scan code of 0x0194 to the end of the table, as shown:

```
const UINT16 device_2_keyboard_scan_table[] =
{
    0x0192, // Calculator
    0x0223, // WWW Home
    0x00CD, // Play/Pause
    0x0221, // WWW Search
    0x018A, // Mail
    0x00E9, // Volume Up
    0x00E2, // Mute
    0x00B7, // Stop
    0x00EA, // Volume Down
    0x022A, // WWW Favorites
    0x0225, // WWW Forward
    0x0224, // WWW Back
    0x00B6, // Scan Previous Track
    0x00B5, // Scan Next Track
    0x0194, // My Computer
};
```

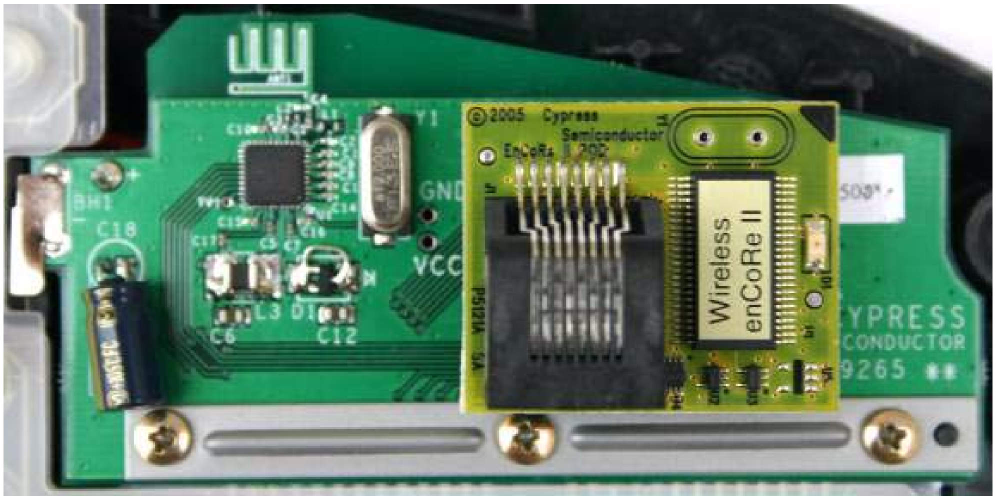
3. Build the firmware, the new key **My Computer** will work.

4.5 Development Environment

4.5.1 Tools

- See the *CY4636 Getting Started Guide* for a list of tools required to build and debug the keyboard application.

Figure 4-9. Figure 20: RDK Keyboard with POD Installed



4.5.2 Tips and Tricks

4.5.2.1 M8C Sleep

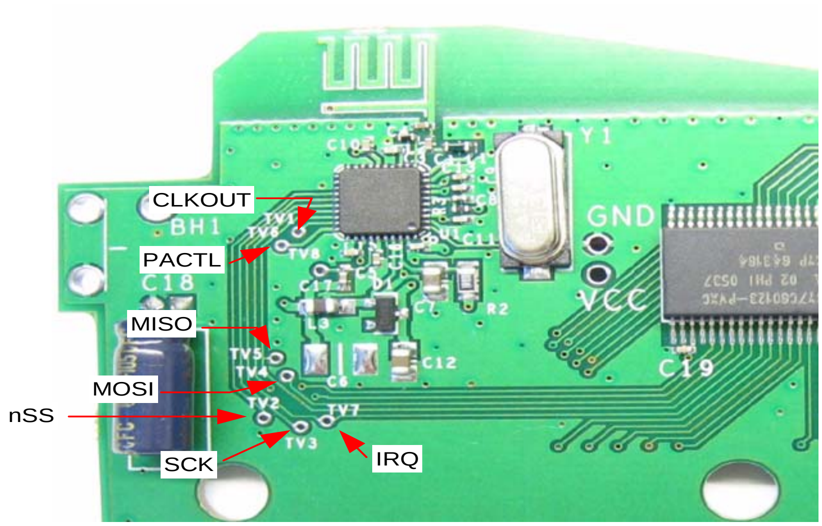
When using the ICE-Cube, define the macro PSOC_ICE so that busy waits are used instead of the sleep instruction. Using the sleep instruction with the ICE-Cube generates errors due to synchronization issues between the OCD part and the emulator.

4.5.2.2 Watchdog Timer

The watchdog timer is enabled for the RDK operation, but may be disable for debug purposes.

4.5.3 Critical Test Points

Figure 4-10. RDK Keyboard Test Points



[+] Feedback

5. Bridge

5.1 Introduction

This section covers the design goals, architecture, firmware source code modules and configuration options for the WirelessUSB LP bridge. It does not cover the details of the radio subsystem or the configuration options that go with it.

5.1.1 Design Features

The CY4636 Reference Design Kit uses an enCoRe II controller for the LP RDK bridge (Cypress part #: CY7C63803-SXC). Contact your local sales representative for more information on the enCoRe II controller.

The architecture was designed to be modular for extendibility and maintainability. It was also designed so that it could easily be ported from one hardware platform to another assuming the use of an enCoRe II microprocessor with USB hardware support. Porting to another microprocessor requires more work to account for the USB hardware support and other hardware specific changes.

Design efforts have been made to reduce the ON time of the microprocessor and radio to conserve battery life of attached devices. This includes protocol optimizations along with using sleep features of the radio and enCoRe II microprocessor.

5.2 Hardware Overview

The WirelessUSB LP HID bridge is provided with the RDK. This bridge may be plugged into the USB port on a PC to provide the Wireless USB bridge functionality. The bridge firmware runs on an enCoRe™ II CY7C63803-SXC chip, is written in C and assembly, and runs on the PDC-9263 USB HID bridge. The rest of this section gives a functional overview of the bridge firmware.

The bridge connects the remote WirelessUSB LP HIDs to a low-speed USB host. This firmware supports 2-way communication with bridge and HID devices configured as transceivers.

Packets similar to Standard USB HID packets are encapsulated inside WirelessUSB LP packets, which also contain a packet header and CRC to help the bridge correctly process the USB HID data packets. Valid packets are then sent via USB to the USB host.

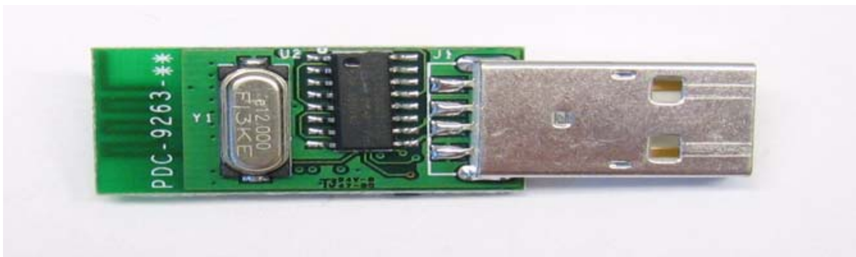
5.2.1 Bridge Photographs

Figure 5-1 shows the top side of the RDK bridge board. The side button on the board is the 'Bind' button.

Figure 5-1. RDK Bridge Top

[Figure 5-2](#) shows the bottom side of the RDK bridge board.

Figure 5-2. RDK Bridge Bottom



5.2.1.1 *In-System Programming*

The LP RDK Bridge has the capability of being programmed through the USB connector using a Cypress USB adapter board PDC-9241 as shown in Figure 5-3.

Figure 5-3. Cypress USB Programming Adapter

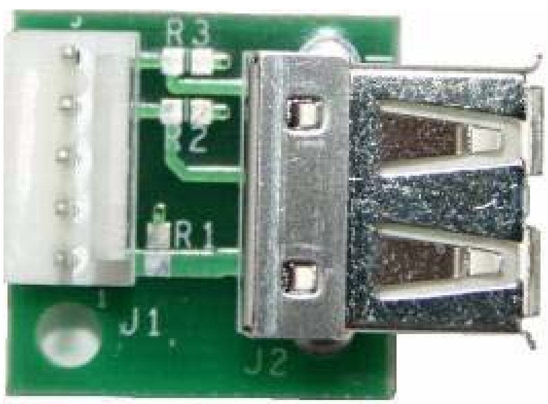


Figure 5-4 on page 65 shows the LP RDK bridge connected with an USB adapter board to a PSoC MiniProg.

Figure 5-4. RDK Bridge with USB Adapter and PSoC MiniProg

5.2.2 Schematics

The LP RDK bridge schematics and Gerber files are located in the following directory: `<installation directory>\Hardware\bridge`. The schematic is in Adobe Acrobat PDF format with the letters 'Sch' in the file name.

5.2.3 LED Usage

Red LED:

- When the dongle is first plugged in, the red LED turns on. It turns off when the USB enumeration process completes. **Note** If the Bind button is pushed when the dongle is first plugged in, the firmware enters MTK test mode and blinks the LED. The LED blinks continuously until the dongle is removed from the computer.
- The red LED blinks ON/OFF when the dongle is in Bind Mode. The ON and OFF time is approximately 320 ms which is the rate at which the dongle changes channels during the Bind process.
- The red LED also blinks ON/OFF when the PC is suspended. The blinking rate is approximately 1 second which is the frequency of the wake up interrupts.

Green LED:

- The green LED turns ON when the dongle receives data from mouse or keyboard. It remains ON for 250 ms after the last received Data packet.
- The green LED turns on and remains on if a key is pressed and held (due to the keyboard's sending Keep Alive packets).
- The green LED turns ON and remains ON during Ping Mode (in normal operation, Ping Mode is a very short period. The user may not notice this period).

The Red and Green LED are blinking alternately when in Manufacturing Test mode.

5.3 Firmware Architecture

There are two architectural views of the bridge. The first is a microcontroller configuration view of User Modules mapped to digital and analog blocks inside the controller. This architecture and configuration is best viewed in the PSoC Designer application when the project is loaded. The second view is a logical organization of the source code modules that make up the bridge application code and other support modules.

The next two sections describe both architectures with emphasis on top-level organization and overall module operation. More detailed description of variables and functions should be obtained by referencing the source code.

5.3.1 ROM/RAM Usage

The following table shows the ROM/RAM usage. The top part exhibits the total ROM/RAM usage for basic functions, which disables all the build options below. The bottom part exhibits the ROM/RAM usage for individual build options.

Table 5-1. ROM/RAM Usage

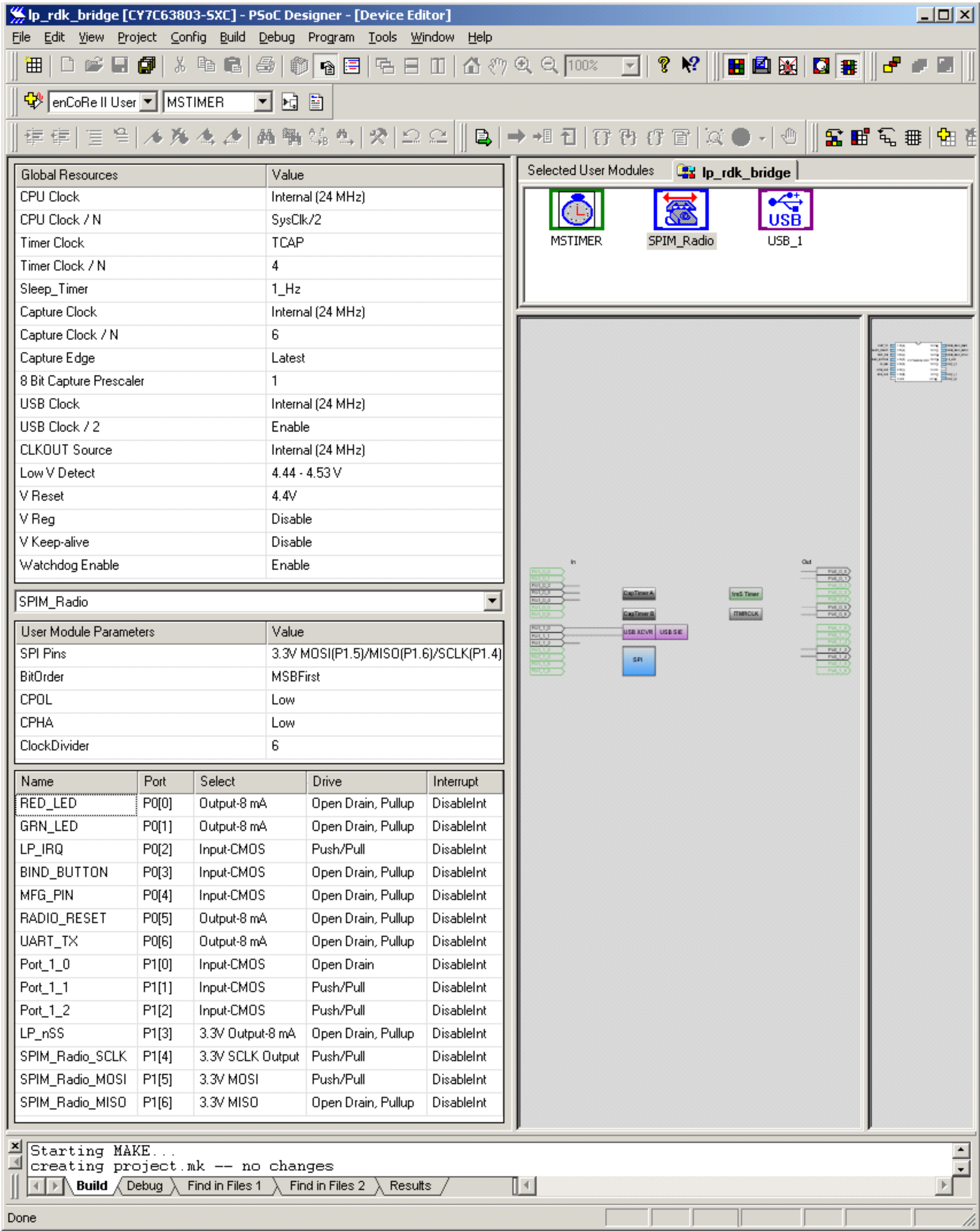
	Total ROM (Bytes)	**Total RAM (Bytes)**
Basic Functions	6522	164
Build Option	**ROM Usage (Bytes)**	**RAM Usage (Bytes)**
ENCRYPT_DATA*	887	29
MFG_TEST_CODE	465	0
MFG_TX_MODES	632	2

*The ENCRYPT_DATA option requires 64 bytes of extra ROM space to store the non-volatile session key.

5.3.2 enCoRe II Device Configuration

The enCoRe II is configured using the Device Editor in PSoC Designer. The bridge uses the SPI Master, USB Device, and the 1 Millisecond Interval Timer enCoRe User Modules. The SPI Master User Module is used by firmware to communicate with the LP radio. The USB Device User Module allows the bridge to operate as a low-speed USB device. The 1 Millisecond Interval Timer User Module is used for timing. Following is a screen shot of the Device Editor showing the User Module mapping. Further description of resources and User Modules follow the diagram.

Figure 5-5. Microcontroller Device Architecture



5.3.2.1 *Global Configuration*

The following is a description of the **Global Resources** that are configured for the CY7C63803-SXC enCoRe II microcontroller. Care must be taken when modifying these values as they affect the User Modules discussed below.

5.3.2.1.1 CPU Clock

This parameter is set to Internal (24 MHz). In order to run the CPU at 12 MHz CPU Clock/N must be set to 2. This operating frequency provides for faster code execution.

5.3.2.1.2 CPU Clock / N

This parameter is set to 2 to provide a 12 MHz clock.

5.3.2.1.3 Timer Clock

This parameter is set to TCAP.

5.3.2.1.4 Timer Clock /N

This parameter is set to 4.

5.3.2.1.5 Capture Clock

This parameter is set to Internal (24MHz).

5.3.2.1.6 Capture Clock /N

This parameter is set to 6.

5.3.2.1.7 Capture Edge

This parameter is set to Latest.

5.3.2.1.8 8 Bit Capture Prescaler

This parameter is set to 1.

5.3.2.1.9 USB Clock

This parameter is set to Internal (24 MHz).

5.3.2.1.10 USB Clock /2

This parameter is set to Enable.

5.3.2.1.11 CLKOUT Source

This parameter is set to Internal (24 MHz).

5.3.2.1.12 Low V Detect

This parameter is set to 4.44-4.53 V.

5.3.2.1.13 V Reset

This parameter is set to 4.4V.

5.3.2.1.14 VReg

This parameter is set to 'Disable' here. The VReg is enabled in the application code to implement the hard reset to the LP radio.

5.3.2.1.15 V Keep-alive

This parameter is set to Disable.

5.3.2.1.16 Watchdog Enable

This parameter should be set to Enable, but may be set to Disable for debug purposes.

5.3.2.2 SPI Master User Module

The SPI Master User Module is used by firmware to communicate with the LP radio. The interrupt API to this module is not used.

5.3.2.3 USB Device User Module

The USB Device User Module handles the enumeration and data transfers over USB endpoints.

5.3.2.4 1 Millisecond Interval Timer User Module

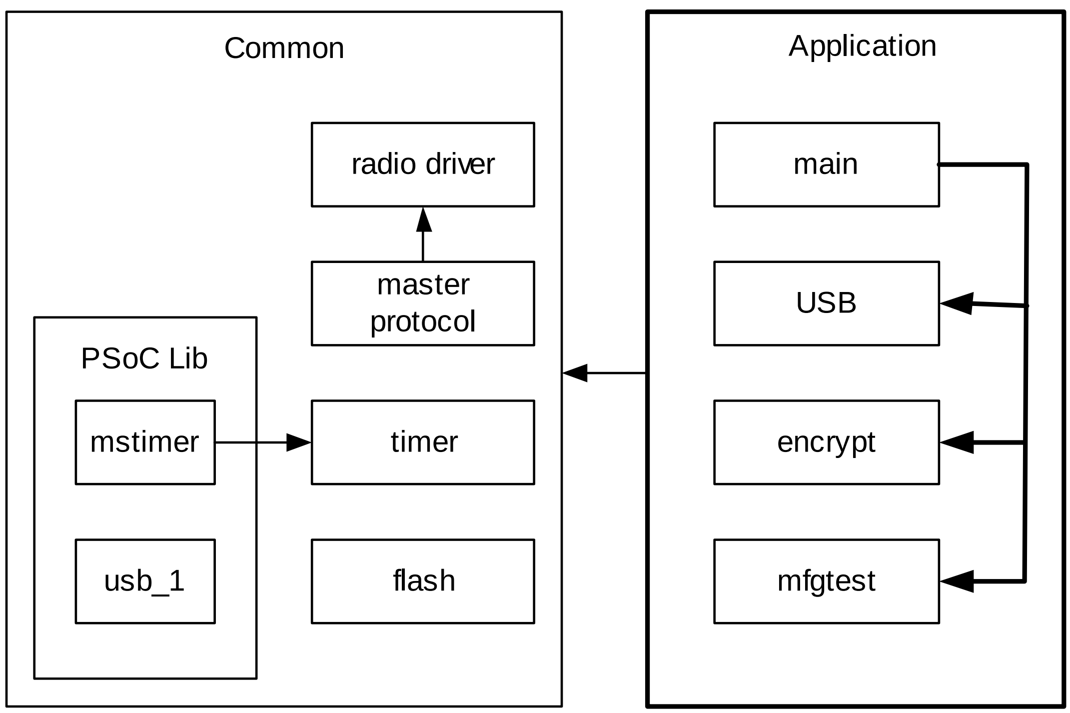
The 1 Millisecond Interval Timer User Module is used to determine when a USB suspend has occurred, LED on/off duration timing, RSSI checking and others.

5.3.2.5 Flash Security

The bridge project within PSoC Designer has a file called *FlashSecurity.txt*. This file specifies access rules to blocks of the Flash ROM. Refer to the documentation listed at the top of the file for definitions. This file is shipped with a single change from its default configuration. The block starting at address 0x1FC0 has been changed from W: Full (Write protected) to U: Unprotected. This location of Flash has been dedicated to saving non-volatile session key for the **encrypt** code module.

5.3.3 Model

Figure 5-6. Firmware Architecture Model



The bridge firmware is partitioned into two logical groups. The Common group is a collection of code modules that provide the underlying support for the application. This group provides services such as, radio protocol, radio driver, USB, timing, flash access, SPI and interrupts.

The Application group implements the core functionality and features of RDK wireless bridge. This includes USB HID packet formatting and reporting, encryption, and manufacturing test mode. The code modules for each of these groups are described below in further detail.

All of the following module descriptions have corresponding <module name>.c and <module name>.h source code files. The module API and definitions are exported in the header file while the module implementation and local definitions are contained in the C file.

5.3.4 Common Code

The modules consist of the common code logical grouping.

5.3.4.1 PSoC Generated Library Code

There are currently only three files, generated by PSoC Designer, that are modified for the use of the application. A minimal amount of code has been added to these modules in user protected areas that are preserved across code generation.

5.3.4.1.1 USB include (USB_1.inc)

This file includes the additional code for the Battery Level and Link Quality software application in *USB_1_cls_hid.asm*.

5.3.4.1.2 USB HID Class Module (*USB_1_cls_hid.asm*)

The additional user code provides support for the Battery Level and Link Quality software application.

5.3.4.1.3 1 Millisecond Interval Timer Interrupt Module (*MSTIMER.asm*)

The additional user code decrements application countdown timers and checks for USB activity to detect a USB suspend condition.

5.3.4.2 Flash

The module includes routines to write to the enCoRe II flash to Encryption key.

5.3.4.3 Timer

The module includes busy wait time routines.

5.3.4.4 Radio Driver

The radio driver module is a low level module providing basic radio communication and configuration. Its general application is such that it is likely not to be changed by the firmware developer. It provides an interface for reading/writing radio registers, setting PN codes and initialization of the radio and transmitting or receiving packets. See the Radio Driver documentation for details.

5.3.4.5 Master Protocol

The module includes LP RDK master protocol routines to handle ping, button bind, channel agility and data packets. This module has a dependency on the **radio driver** for sending and receiving formatted packets and the **flash** module.

5.3.5 Application Code

The group of modules that make up the application code is responsible for implementing the bridge functionality and behavior.

5.3.5.1 *Bridge Module*

The **bridge** module is the controlling code for the application. It has many responsibilities in implementing various features and functions offered by the bridge. The function *main*() is the entry point for the bridge application. This function is called from the *boot.asm* file. The bridge first initializes all of the application modules and then initializes the **master_protocol** module. There is an order dependency for some of these, so care must be taken in modifying the *bridge_init*() function. For example, other modules depend upon the timer facility running in order to perform initialization. Once each module has been initialized, then the application checks for entry to the manufacturing test mode. If the manufacturing test mode is not indicated, then normal bridge operation begins.

The bridge continuously checks the USB idle timer, received packet, the Bind button and the USB suspend.

5.3.5.1.1 Check the USB Idle Timer

The *check_usb_idle()* function is called within the *main()* function to properly handle the USB Set_Idle command. The USB Set_Idle command from the host PC is used to silence the keyboard or mouse report until a new event occurs or the specified amount of time passes. If the host PC's Set_Idle command sets the Idle Duration to '0', the keyboard or mouse endpoint inhibits reporting forever, only reporting when a change is detected in the report data. This causes the bridge to NAK any polls on the endpoint while its current report remains unchanged. If the Set_Idle command sets the Idle Duration to a non-zero number, a single report is generated by the endpoint if the given time duration elapses with no change in report data (see the HID Specification for more information on this topic).

The *check_usb_idle()* function also checks the time out for down key and 'keep alive' packet. A 'keep alive' packet is transmitted every 65 ms during the time a key is pressed, so that the bridge can detect if the RF link is lost, and in that unlikely case, the bridge inserts a 'key up' event, to prevent a 'stuck key' state being transmitted to the PC. The number of milliseconds before upkey reports are generated is defined by DOWNKEY_TIME_OUT.

5.3.5.1.2 Check the Received Packet

When the bridge receives a valid packet, it parses the packet. If it is a data packet, the bridge formats and sends a USB packet to the USB host. If it is a connect request with an approved device or a ping request, the bridge sends a response correspondingly.

5.3.5.1.3 Check the Bind Button

The bridge checks the Bind button frequently. If this button is pressed, the bridge goes into the bind state.

5.3.5.1.4 Check the USB Suspend

The *check_usb_suspend()* monitors the USB suspend condition on the USB bus and takes proper actions to put the system into a low power state when no bus activity is observed for 3 ms.

When suspended, the bridge supports remote wakeup by intermittently turning the radio on when the sleep timer interrupt occurs, checking for valid data from the HID devices, and then turning the radio off again if no HID traffic was detected.

5.3.5.2 *USB Module*

This module parses the radio packets, builds the appropriate keyboard and mouse USB packets and loads these packets into the endpoints.

5.3.5.3 *Mfgtest Module*

The manufacturing test module may be conditionally compiled in to provide manufacturing test support. The module configures the radio for reception and then enters a loop waiting for command packets to be sent from the tester. The test echoes all echo command packets appended with the number of invalid bits received and all other 'valid' command packets (no invalid bits). The manufacturing test code can only be exited by cycling power. The manufacturing test code in this bridge is compatible with the CY3631 Manufacturing Test Kit offered by Cypress Semiconductor.

The manufacturing test mode on the LP RDK bridge can be entered by three different methods depending on the compile-time configuration.

Method 1: Holding the Bind button during insertion into a USB Host enters the manufacturing test mode.

Method 2: Forcing an SE1 condition (D+ and D– are both high) on the USB bus and applying power to the bridge.

Method 3: Grounding P0.4 during insertion into a USB Host enters the manufacturing test mode.

5.3.5.4 *Encrypt Module*

This module may be conditionally compiled in to provide encryption/decryption support. Encrypted data transfers are typically used between RDK keyboard devices and the RDK bridge. Contact Cypress Applications support for the encryption source code.

5.3.6 Configuration Options

All configuration options for the application can be found in the *config.h* file. Options may be configured by defining/un-defining certain #define statements.

5.3.6.1 *MFG_TEST_CODE*

This configuration definition is used to selectively compile in the manufacturing test code. The manufacturing test code in this bridge is compatible with the CY3631 Manufacturing Test Kit offered by Cypress Semiconductor. See the **mfgtest** module for a description of how this test mode is executed. See the CY3631 Manufacturing Test Kit documentation for a description of the test operation.

5.3.6.2 *MFG_TX_MODES*

When the MFG_TEST_CODE is defined, the definition of this name adds in a carrier and random data TX test option. See the **mfgtest** module for more information on these TX modes.

5.3.6.3 *MFG_ENTER_BY_PIN*

This configuration definition is used to selectively compile in a method to enter the manufacturing test code. When this value is defined, the manufacturing test code may be executed by grounding a specific pin during insertion of the LP RDK bridge into a powered USB port or applying external power.

5.3.6.4 *MFG_ENTER_BY_BUTTON*

This configuration definition is used to selectively compile in a method to enter the manufacturing test code. When this value is defined, the manufacturing test code may be executed by holding the Bind button during insertion of the LP RDK bridge into a powered USB port or applying external power.

5.3.6.5 *MFG_ENTER_BY_USBSE1*

This configuration definition is used to selectively compile in a method to enter the manufacturing test code. When this value is defined, the manufacturing test code may be executed by causing a USB SE1 condition on the D+ and D- signals during insertion of the LP RDK bridge into a powered USB port or applying external power.

5.3.6.6 *ENCRYPT_DATA*

This configuration definition is used to selectively compile in data encryption/decryption. Contact Cypress Applications support for the encryption source code.

5.3.6.7 *GREEN_LED_ON_TIME*

This configuration definition defines the number of milliseconds the Green LED stays on after a valid USB report is loaded in an endpoint.

5.3.6.8 *DOWNKEY_TIME_OUT*

This configuration definition defines the number of milliseconds before upkey reports are generated by the bridge in the absence of valid packets from an attached keyboard device.

5.3.7 Platform and Architecture Portability

The bridge firmware was designed to use the hardware features of the enCoRe II such as USB.

Porting the code to another microprocessor architecture may require modification of the existing code to support the different processor specific features.

5.3.8 Initialization

Initialization of the enCoRe II chip is done by code that is generated in *boot.asm* by the PSoC Designer software. The module *boot.asm* calls main once the enCoRe II has been configured and initialized.

Main initializes the components of the bridge along with the radio modules. The bridge firmware enters a loop to receive and handle radio packets and generate USB packets.

5.3.9 Wireless Protocol Data Payload

The RDK HID protocol has been optimized to reduce the ON time of the radio, which equates to reduced power consumption on the LP devices. Refer to the RDK keyboard and RDK mouse sections for radio packet format details.

5.3.10 Suspend and Remote Wake-up

In order to meet the USBIF Compliance requirements regarding power consumption during suspend state, the WirelessUSB LP RDK bridge must reduce the over all power consumption to less than 500 µA if Remote Wakeup is not enabled (Remote Wakeup is the device ability to wake up a suspended PC with user’s input such as a key press, mouse movement, etc.) Because the WirelessUSB LP RDK is not configured to wake up the suspended host PC, the entire bridge must go into deep sleep state to conserve power. Only bus activity from the host PC can bring the bridge back to normal operation.

If Remote Wakeup is enabled, the bridge may draw up to 2.5 mA in suspend state. This requires that the radio circuitry be off most of the time. It is necessary to periodically turn the radio on to sense activity from the WirelessUSB LP mouse or keyboard (and thereby know when to wake the host). The wake up period is configurable and is set to 1 second (see Register OSC_CR0 setting). Increasing the wakeup interrupt frequency results in a faster response to the user's wakeup events at the expense of a slightly higher than average sleep current.

Table 5-2. Bridge Average Icc in Suspend State

Parameter	Icc	Units
bridge Average Suspend Power Consumption - REMOTE WAKE UP ENABLED	1.44	mA
bridge Average Suspend Power Consumption - REMOTE WAKE UP DISABLED	0.3	mA

5.3.11 Interrupt Usage and Timing

The polling method is used for the Bind button.

5.3.12 Code Performance Analysis

A keyboard report processing is used to analyze the code performance. A typical keyboard report processing contains the following steps:

- The bridge receives the keyboard report packet and process the packet. This step takes 108 µs.
- MCU calls function handle_keyboard_report() to format USB packet and load this packet into the endpoint buffer. This function consumes 118 µs.

As a result, it takes 226 µs for the bridge to process a keyboard report.

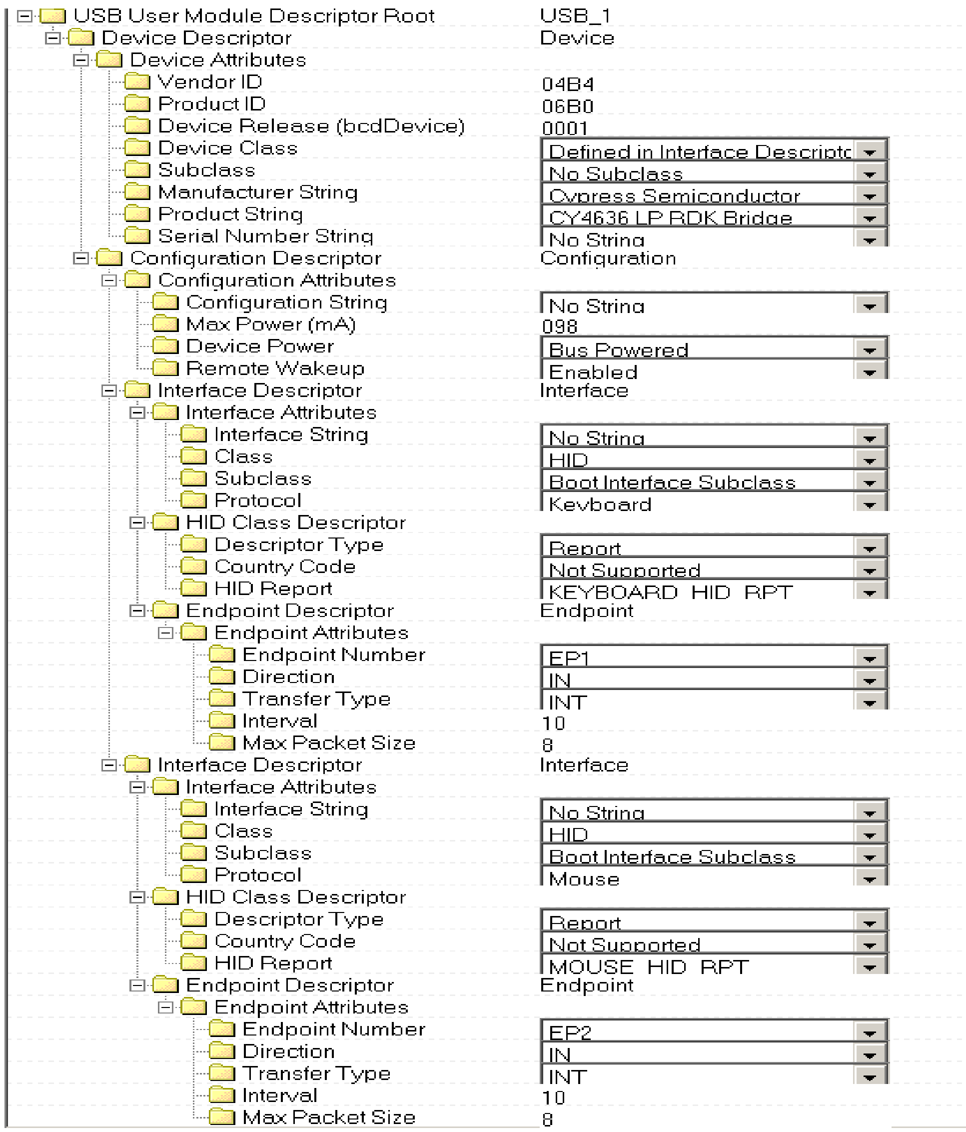
5.4 USB Interface

5.4.1 USB Descriptors

The USB Descriptors can be viewed/edited with the USB Setup Wizard.

5.4.1.1 Device/Config Descriptors

Figure 5-7. USB Device/Config Descriptors



5.4.1.2 *Keyboard HID Report Descriptor*

The keyboard HID report descriptor defines a Boot Protocol keyboard. This enables a LP RDK keyboard with the LP RDK bridge to work on different BIOS versions that do not correctly support the USB Report Protocol. Only standard 101(104) keys are sent using this format over endpoint 1.

Figure 5-8. Keyboard HID Report Descriptor (Endpoint 1)

HID Report Descriptor	KEYBOARD_HID_RPT
Usage Page	Usage Page 05 01
Usage	Usage 09 06
Collection	Collection (Application 01)
Usage Page	Usage Page 05 07
Usage Minimum	Usage Minimum 19 E0
Usage Maximum	Usage Maximum 29 E7
Logical Minimum	Logical Minimum 15 00
Logical Maximum	Logical Maximum 25 01
Report Size	Report Size 75 01
Report Count	Report Count 95 08
Input	Input (Data, Variable, Absolute 02)
Report Size	Report Size 75 08
Report Count	Report Count 95 01
Input	Input (Constant 01)
Usage Page	Usage Page 05 07
Usage Minimum	Usage Minimum 19 00
Usage Maximum	Usage Maximum 29 A4
Logical Maximum	Logical Maximum 26 A4 00
Report Count	Report Count 95 06
Input	Input (Data, Array, Absolute 00)
End Collection	End Collection C0

5.4.1.3 *Mouse/Keyboard HID Report Descriptor*

The mouse/keyboard HID Report Descriptor uses report protocol format with a unique report ID for each report. Mouse data uses Report ID 1. The mouse report include delta x, delta y, and scroll wheel data.

Figure 5-9. Mouse HID Report Descriptor (Report ID 1 – Endpoint 2)

HID Report Descriptor Root	USB_1
HID Report Descriptor	KEYBOARD HID RPT
HID Report Descriptor	MOUSE HID RPT
Usage Page	Usage Page 05 01
Usage	Usage 09 02
Collection	Collection (Application 01)
Usage	Usage 09 01
Collection	Collection (Physical 00)
Report ID	Report ID 85 01
Report Count	Report Count 95 03
Report Size	Report Size 75 01
Usage Page	Usage Page 05 09
Usage Minimum	Usage Minimum 19 01
Usage Maximum	Usage Maximum 29 03
Logical Minimum	Logical Minimum 15 00
Logical Maximum	Logical Maximum 25 01
Input	Input (Data, Variable, Absolute 02)
Report Count	Report Count 95 01
Report Size	Report Size 75 05
Input	Input (Constant 01)
Report Size	Report Size 75 08
Report Count	Report Count 95 02
Usage Page	Usage Page 05 01
Usage	Usage 09 30
Usage	Usage 09 31
Logical Minimum	Logical Minimum 15 80
Logical Maximum	Logical Maximum 25 7F
Input	Input (Data, Variable, Relative 06)
End Collection	End Collection C0
Collection	Collection (Physical 00)
Report Count	Report Count 95 01
Report Size	Report Size 75 08
Usage Page	Usage Page 05 01
Usage	Usage 09 38
Logical Minimum	Logical Minimum 15 81
Logical Maximum	Logical Maximum 25 7F
Input	Input (Data, Variable, Relative 06)
End Collection	End Collection C0
End Collection	End Collection C0

Keyboard multimedia keys use Report ID 2.

Figure 5-10. Keyboard's MM Keys HID Report Descriptor (Report ID 2 – Endpoint 2)

Usage Page	Usage Page 05 0C
Usage	Usage 09 01
Collection	Collection (Application 01)
Report ID	Report ID 85 02
Usage Minimum	Usage Minimum 19 00
Usage Maximum	Usage Maximum 2A 3C 02
Logical Minimum	Logical Minimum 15 00
Logical Maximum	Logical Maximum 26 3C 02
Report Count	Report Count 95 01
Report Size	Report Size 75 10
Input	Input (Data, Array, Absolute 00)
End Collection	End Collection C0

Keyboard power keys use Report ID 3.

Figure 5-11. Keyboard's Power Keys HID Report Descriptor (Report ID 3 – Endpoint 2)

Usage Page	Usage Page 05 01
Usage	Usage 09 80
Collection	Collection (Application 01)
Report ID	Report ID 85 03
Usage Minimum	Usage Minimum 19 81
Usage Maximum	Usage Maximum 29 83
Logical Minimum	Logical Minimum 15 00
Logical Maximum	Logical Maximum 25 01
Report Size	Report Size 75 01
Report Count	Report Count 95 03
Input	Input (Data, Variable, Absolute 02)
Report Count	Report Count 95 05
Input	Input (Constant 01)
End Collection	End Collection C0

Report ID 4 is used to send the mouse battery level and link quality report.

Figure 5-12. Mouse's Battery/Link Quality Report Descriptor (Report ID 4 – Endpoint 2)

Usage Page	Usage Page 06 01 FF
Usage	Usage 09 02
Collection	Collection (Application 01)
Report ID	Report ID 85 04
Report Count	Report Count 95 01
Report Size	Report Size 75 08
Logical Minimum	Logical Minimum 15 01
Logical Maximum	Logical Maximum 25 0A
Usage	Usage 09 20
Feature	Feature (Constant 03)
Logical Maximum	Logical Maximum 25 4F
Usage	Usage 09 21
Feature	Feature (Constant 03)
Logical Maximum	Logical Maximum 25 30
Usage	Usage 09 22
Feature	Feature (Constant 03)
Report Size	Report Size 75 10
Usage	Usage 09 23
Feature	Feature (Constant 03)
Usage	Usage 09 24
Feature	Feature (Constant 03)
End Collection	End Collection C0

Report ID 5 is used to send the keyboard battery level and link quality report.

Figure 5-13. Keyboard's Battery/Link Quality Report Descriptor (Report ID 5–Endpoint 2)

Usage Page	Usage Page 06 01 FF
Usage	Usage 09 01
Collection	Collection (Application 01)
Report ID	Report ID 85 05
Report Count	Report Count 95 01
Report Size	Report Size 75 08
Logical Minimum	Logical Minimum 15 01
Logical Maximum	Logical Maximum 25 0A
Usage	Usage 09 20
Feature	Feature (Constant 03)
Logical Maximum	Logical Maximum 25 4F
Usage	Usage 09 21
Feature	Feature (Constant 03)
Logical Maximum	Logical Maximum 25 30
Usage	Usage 09 22
Feature	Feature (Constant 03)
Report Size	Report Size 75 10
Usage	Usage 09 23
Feature	Feature (Constant 03)
Usage	Usage 09 24
Feature	Feature (Constant 03)
End Collection	End Collection C0

5.4.2 Keyboard Report Format

The keyboard standard keys information is sent to the host PC via the data endpoint 1. The keyboard multimedia keys and power keys information is sent to the host PC via the data endpoint 2 using Report ID (the first byte in the report). The mouse uses Report ID 1. The keyboard multimedia keys use Report ID 2. The keyboard power keys use Report ID 3. The formats of the keyboard report are shown below:

Figure 5-14. Keyboard Report Format

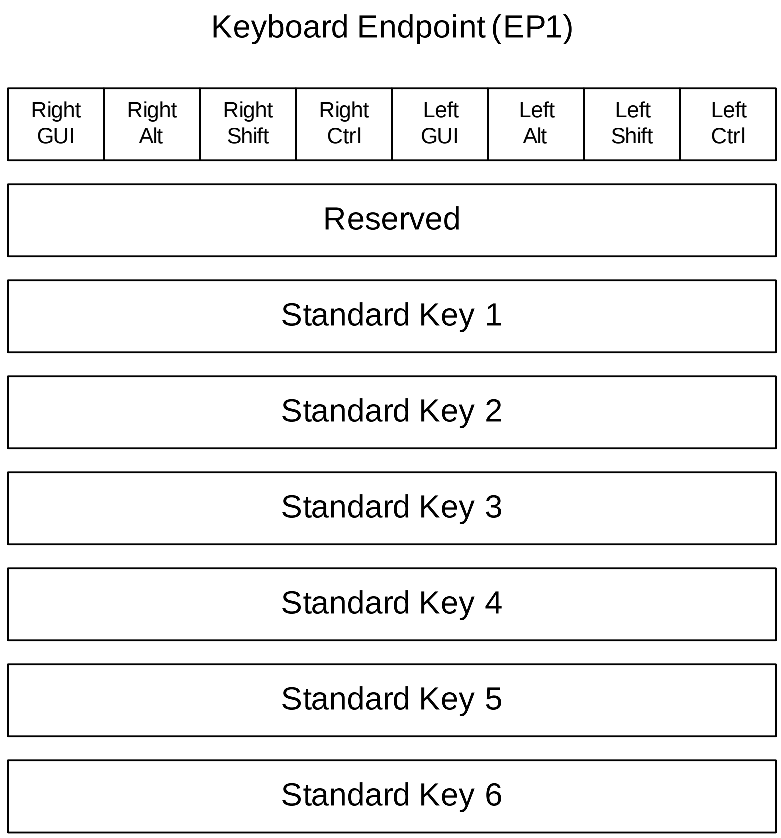
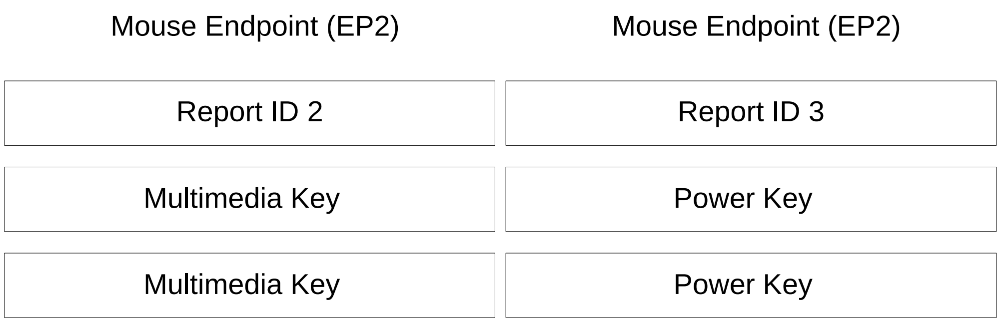


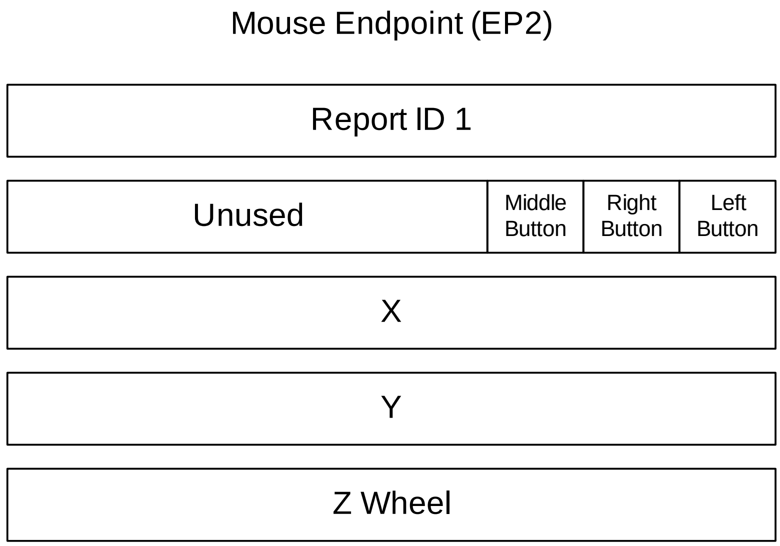
Figure 5-15. Multimedia and Power Keys Report Format



5.4.3 Mouse Report Format

The mouse data is sent over the data endpoint 2 using Report ID 1. The format of the mouse report is shown below:

Figure 5-16. Mouse Report Format



5.4.4 Battery Level and Link Quality Reports

The WirelessUSB LP bridge implements a mechanism to report the radio parameters of attached HID devices via the USB control endpoint. The code for this functionality can be found in the user custom code section of the User Module source file *usb_1_cls_hid.asm*.

The RadioParams HID report is a vendor-defined HID report for communicating several radio parameters of the WirelessUSB LP HID devices.

The HID Report Page is defined as:

Cypress WirelessUSB HID RadioParams Report Page (0xFF01 – Vendor Defined)

Table 5-3. USB Report Usage IDs

Usage ID	Usage Name
0x00	Undefined
0x01	WirelessUSB keyboard
0x 02	WirelessUSB mouse
0x03-0x1F	RESERVED
0x 20	Battery Level
0x 21	WirelessUSB Channel
0x 22	WirelessUSB PN Code
0x 23	Corrupt Packets
0x 24	Packets Transferred

The RadioParams Report is 8 bytes long and has the 6 data fields listed in Table 5-4.

Table 5-4. USB Report Format

Byte	Use	Range
0	Report ID #	0x04
1	Battery Level	0 – 0x0A
2	Channel #	0 – 0x4D
3	PN Code	0 – 0x30
4-5	Corrupt Packets	0 – 0xFFFF
6-7	Packets Transferred	0 – 0xFFFF

5.4.4.1 *Requesting a New Battery Reading*

When the Bridge receives a control endpoint request from the host with the following parameters, it returns an 8-byte RadioParams report over the control endpoint. An attached LP device should send an updated battery report whenever a reconnect or a change in the battery level occurs.

Control endpoint request for new battery reading.

Table 5-5. USB Set Report

	Value
bmRequestType	0x21 (To Device, Type = Class, Recipient = Interface)
Request Code	0x09 (**Set Report**)
wValue	0x0304 (Feature Report, ReportID = 4)
wIndex	0x0000 = Kbd, 0x0001 = mouse

5.4.4.2 *Obtaining the RadioParams Report*

When the bridge receives, from the host, a control endpoint request with the parameters listed on Table 5-6, it returns an 8-byte RadioParams report over the control endpoint.

Control endpoint request for RadioParams report are listed.

Table 5-6. USB Get Report

	Value
bmRequestType	0xA1 (From Device, Type = Class, Recipient = Interface)
Request Code	0x01 (**Get Report**)
wValue	0x0304 (Feature Report, ReportID = 4)
wIndex	0x0000 = Kbd, 0x0001 = mouse

When the bridge receives the ***Get Report*** control request code, it returns a RadioParams report and then resets the *Packets Transferred* parameter for the specified device to zero.

The *Link Quality* value is updated whenever the bridge receives a radio packet from the wireless device.

Battery Level is only updated when the device sends an updated battery level report.

At startup, the Battery Level, Corrupt Packets and Packets Transferred are initialized to zero.

5.4.5 Example USB Bus Analyzer (CATC) Traces

Figure 5-17 below shows the USB data transmissions between the bridge and the host PC captured with the USB CATC Bus Analyzer. In this example, the Right Shift + ‘g’, ‘h’ keys were typed followed by the ‘Volume Up’, ‘Volume Down’ keys. Note the keyboard regular key reports are sent to the PC via the endpoint 1 while the Multi Media key reports are sent via the endpoint 2 with Report ID 2.

Figure 5-17. Example keyboard CATC Trace (Standard and MM Keys)

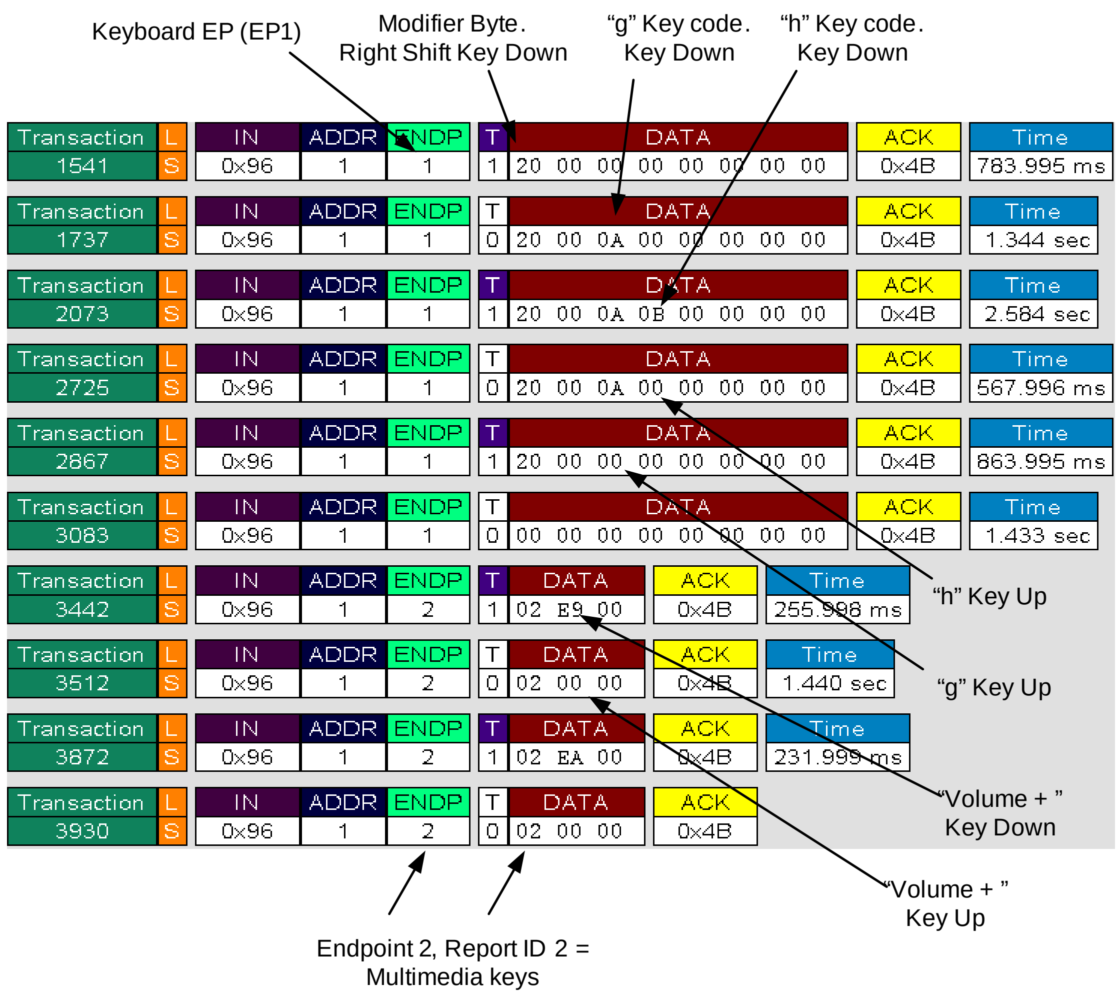


Figure 5-18 below shows the mouse data being transferred between the dongle and the host PC. The first part of the trace shows the mouse data when the left button was pressed and held down as the mouse was moved, and then the left button was released. The second part of the trace shows the Z-wheel being moved down and up.

Figure 5-18. Example Mouse CATC Trace

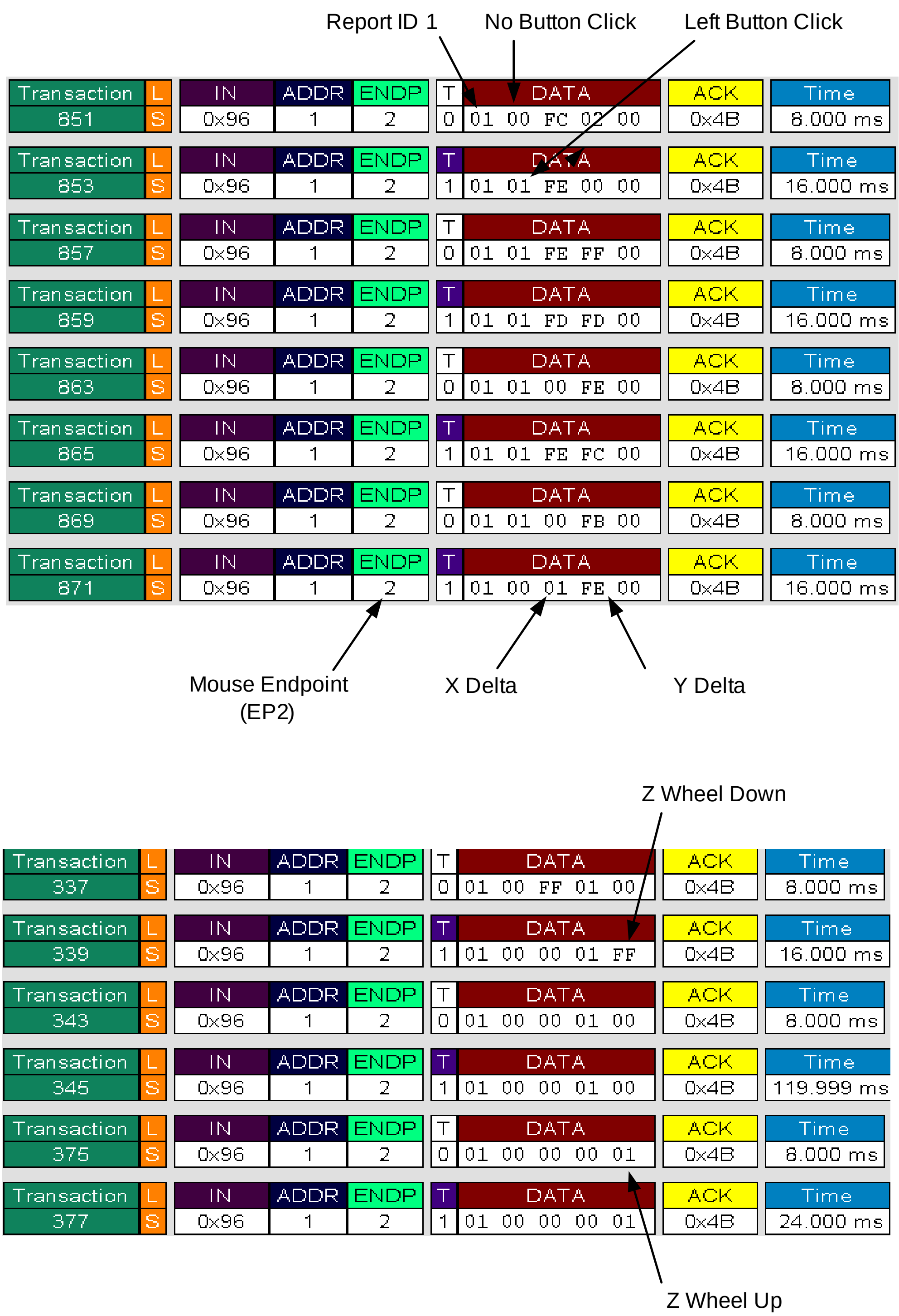


Figure 5-19 shows the Sleep key being pressed. Note the power key reports are sent via endpoint 2 and Report ID 3.

Figure 5-19. Example Keyboard CATC Trace (Power Key)

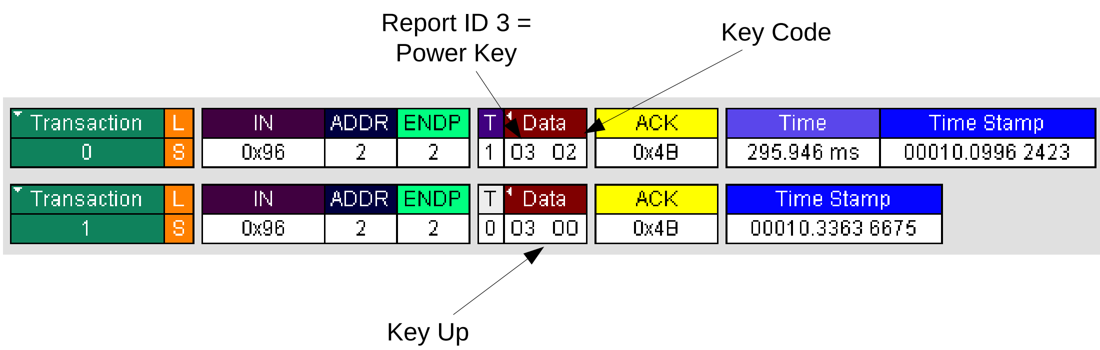
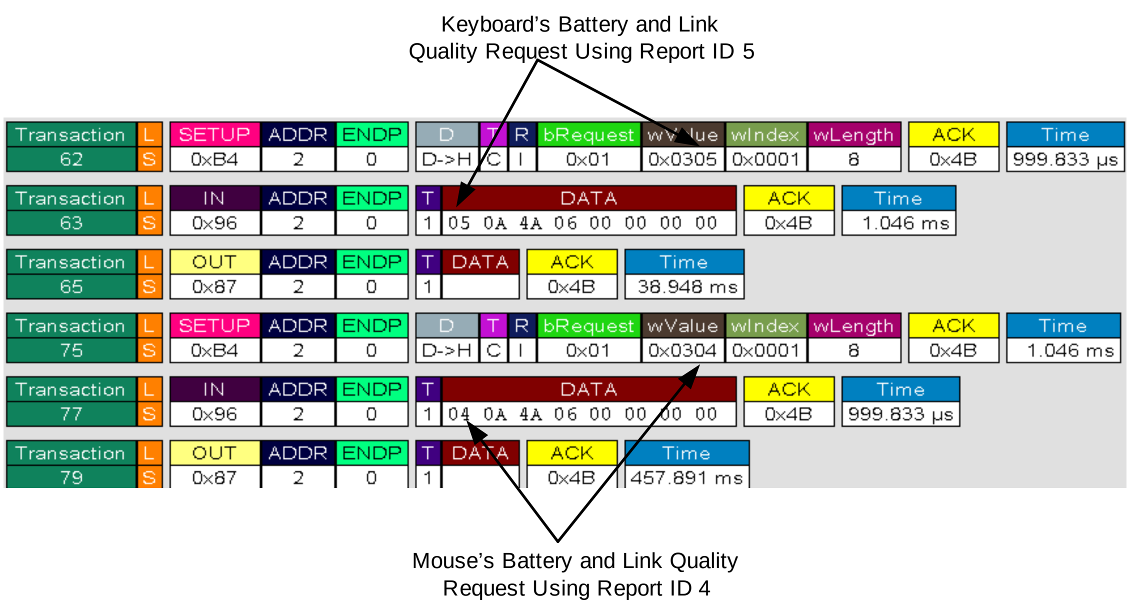


Figure 5-20 below shows the Get_Report requests used to retrieve the keyboard and mouse battery level and link quality information. Note the data transfers occurred on the control endpoint, endpoint 0, and Report ID were used to differentiate keyboard and mouse requests.

Figure 5-20. CATC Trace of Battery and Link Quality Data Requests



5.5 Development and Debug Environment

Information on the tools required and tips on using those tools are presented in this section.

5.5.1 Tools

See the *CY4636 Getting Started Guide* for a list of tools required to build and debug the keyboard application.

Figure 5-21. RDK Bridge with POD Foot

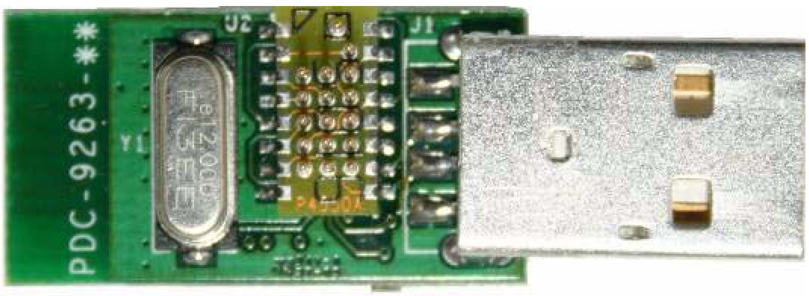
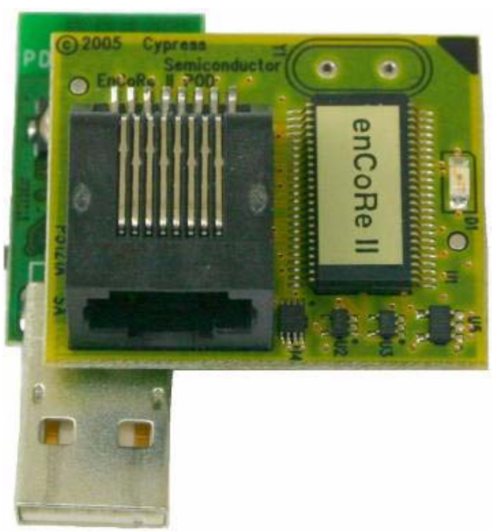


Figure 5-22. RDK Bridge with POD Installed



5.5.2 Tips and Tricks

M8C Sleep

- When using the ICE-Cube, define the macro PSOC_ICE so that busy waits are used instead of the sleep instruction. Using the sleep instruction with the ICE-Cube generates errors due to synchronization issues between the OCD part and the emulator.

Watchdog Timer

The watchdog timer is enabled for the RDK operation, but may be disable for debug purposes.

POD Power

On the Project Settings->Debugger window select ‘Pod uses external power only’ when connected to USB. The other option is to disconnect the VBUS signal on the PCB.

6. Manufacturing Test Support, MTK

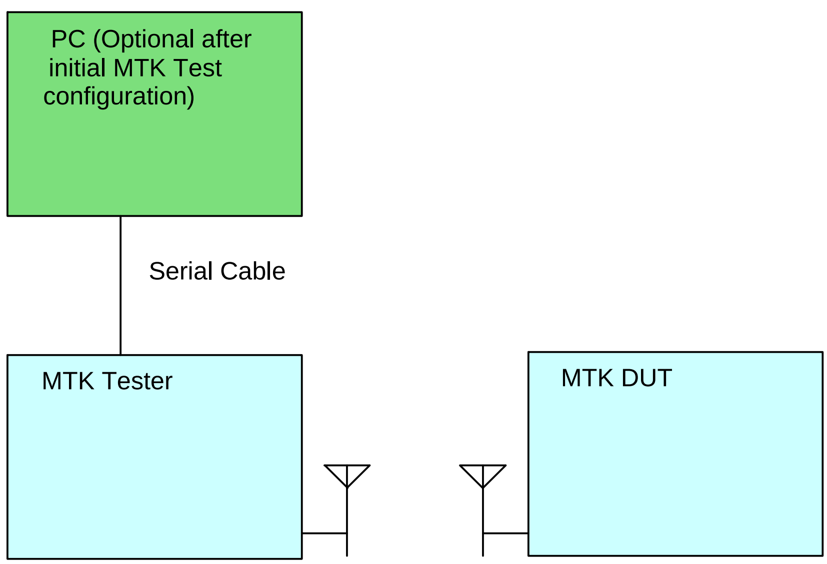
6.1 Introduction

The Manufacturing Test Kit (MTK) provides production line test support in addition to providing FCC certification tests. This section provides a description of the Tester serial protocol, the RF protocol between the MTK Tester and the MTK Device-Under-Test (DUT) and a brief description of porting the MTK DUT code to different platforms.

Refer to the Manufacturing Test Kit User's Guide for instructions on operating the MTK Tester.

6.2 MTK Block Diagram

Figure 6-1. Block Diagram



6.3 MTK Serial Protocol

The MTK Tester implements a text-based protocol over an RS232 serial port to provide both a configurable standard test and script-based testing.

All commands listed under the standard test set a configuration value that is stored in non-volatile storage. All remaining serial commands only affect the current setting and are not stored (reset across power cycles). Table 6-1 on page 88 describes the serial port protocol in the PC to tester direction.

Table 6-1. Serial Command Protocol

Command		Command Description
ST		START STANDARD TEST
Standard Test Parameters	CL <power level>	CONFIGURE POWER LEVEL (0-7)
	PN <PN code>	CONFIGURE PN CODE INDEX (0-7)
	TT <tx error threshold>	CONFIGURE TX ERROR THRESHOLD (0-65535) units of bit errors
	RT <rx error threshold>	CONFIGURE RX ERROR THRESHOLD (0-65535) units of bit errors
	C1 <channel>	CONFIGURE CHANNEL (0-77)
	C2 <channel>	CONFIGURE SECOND CHANNEL (0-77)
	C3 <channel>	CONFIGURE THIRD CHANNEL (0-77)
	CB <# of bytes>	CONFIGURE NUMBER OF BYTES/PACKET PAYLOAD (0-15)
	CP <# of packets>	CONFIGURE NUMBER OF PACKETS (0-255)
TC <time>		TRANSMIT CARRIER (0-255)
TR <time>		TRANSMIT RANDOM (0-255)
SC <channel> <PN code> <power level> <correlator threshold>		SET COMMUNICATION (0-77) (0-7) (0-7) (0-16) Note: The device transmits on <channel> + 2. For example, 2=2.402 GHz
PD <packet data>		SET PACKET DATA (ASCII representation of hexadecimal numbers without any prefix, i.e. 5A 34 CB)
CA <crystal adjust>		SET CRYSTAL FREQUENCY ADJUST VALUE (0-63)
RE		RESTORE NVRAM DEFAULTS
CS		SHOW CURRENT CONFIGURATION
HE		SHOW HELP MENU

Every serial command issued by the PC is returned with a response once the command is complete. The valid responses are shown in Table 6-2.

Table 6-2. Serial Response Protocol

REPORT	REPORT DESCRIPTION
OK	COMMAND COMPLETE
CE	COMMAND ERROR
TE <transmit error count>	TX ERROR COUNT (units of bit errors)
RE <receive error count>	RX ERROR COUNT (units of bit errors)

All serial commands must end in either a carriage return or carriage return and line feed. All responses end with a carriage return and linefeed.

The serial port settings for the MTK Tester are shown in Table 6-3 on page 89. Neither software nor hardware handshake is supported.

Table 6-3. Serial Port Parameter Settings

Serial Port Parameter	Setting
Baud Rate	9600
Parity	None
Number of Data Bits	8
Number of Stop Bits	1

6.4 MTK RF Protocol

Command packets received by the Device-Under-Test (DUT) are ‘echoed’ with the addition of an added byte that contains the count of invalid bits for the received packet. Extra bytes in packets that are larger than what the DUT can support are ignored. Commands other than ‘Echo Packet’ are only ‘echoed’ and executed if the number of invalid bits are zero.

The RF command packets exchanged between the MTK Tester and the MTK DUT contain two bytes. The first byte contains the command type and the subsequent byte(s) contain the parameter values as shown in Table 6-4.

Table 6-4. RF Commands

Description	Command	Parameter
Echo Packet	0x00	N/A
Set New Configuration	0x61	Channel (0-77) PN code index (0-7) PA (0-7) Correlator Threshold (0-16)
Transmit carrier	0x66	Time in seconds (0-255) Note: A zero runs the test continuously until a reset.
Transmit random pattern	0xA3	Time in seconds (0-255) Note: A zero runs the test continuously until a reset.

The ‘Transmit carrier’ and ‘Transmit random pattern’ test mode can be conditionally compiled with the define MFG_TX_MODES.

6.5 MTK DUT source Code Porting

The RDK keyboard, bridge and mouse use the C source files *mfgtest.c* and *mfgtest.h*. Select the appropriate source files for the target platform as a starting point. Make code changes as necessary to work in your environment.

6.6 Accessing MTK in the DUT

Mouse: Apply a jumper across the ISSP header pins 4/5 and install the batteries.

Keyboard: Same as mouse.

Bridge: Press the button while plugging it into the USB port. The LEDs should blink.

7. Regulatory Testing Results

7.1 Introduction

The LP mouse was tested in a certified lab and meets FCC part 15, Subpart B, Title 47 CFR -Unintentional radiators, FCC Part 15 Subpart C -Intentional radiators, and Industry Canada RSS-Gen. The following table outlines the results of the testing.

Testing for the keyboard and bridge is expected in the near future.

Table 7-1. EMC Test Results

Test Parameter	FCC Limit	Measured Value	Margin
Spurious Radiated Emissions	54 dBuV/m(Av)	50.5 dBuV/m	-3.5 dB
Spurious Conducted Emissions	-20dBc	-31.9 dBc	11.9 dB
Power Spectral Density	8 cBm/3 kHz	-9.0 dBm/3 kHz	17.0 dB
Output Power	30 dBm	2.3 mW	26.0 dB
Occupied Bandwidth	>500 kHz	960 kHz	460 kHz
Conducted Band Edge Compliance	20 dB below fundamental	-27.0 dBc	7.0 dB
Radiated Band Edge Compliance at 2482 Mhz	54 dBuV/m(Av)	51.7 dBuV/m	-2.3 dB
	Industry Canada limit		
Receiver Radiated Emissions	46 dBuV/m(QP)	35.4 dBuV/m	-10.6 dB

8. Power Considerations

8.1 RDK Keyboard

8.1.1 Usage Model

The following usage model are considered for the RDK keyboard.

- 4 hours per day of 6 keystrokes per second, 5 days per week
- 24 hours per day with no activity, 2 days per week
- A packet is transmitted on both key-up and key-down events
- A ‘keep alive’ is transmitted for each key-down event

8.1.2 Current Measurements

Per the keyboard usage model, there are 6 keystrokes per second in the active state, and every keystroke includes one ‘down key’ packet, one ‘up key’ packet and one ‘keep alive’ packet. The test mode firmware only sends out one ‘down key’ packet and one ‘up key’ packet for each keystroke. Therefore, we need to set the typing rate to 8 keystrokes per second in test mode in order to consume the equivalent power of the usage model. It is accomplished by changing the KEYBOARD_TEST_MODE_PERIOD define in the *config.h* file to 50.

The following is the results of RDK keyboard current measurement:

Table 8-1. Keyboard Current Measurement

Operation Mode	Icc (mA) with Supply Voltage = 2.5 V	Icc (mA) with Supply Voltage = 2.8 V	Average Icc (mA)
Active state - Place the keyboard in test mode "the quick brown fox …" and set the typing rate to 8 keystrokes per second.	0.96	0.83	0.90
Idle state - A keyboard is in its normal power on state and connected to the bridge with no keys pressed.	0.04	0.04	0.04
Not connected state - Type the keyboard.	20.3	16.9	18.6
Not connected state - No typing.	Transition to idle state.	Transition to idle state.	

8.1.3 Battery Life Calculations

The following table shows the times spent in each state by the RDK keyboard usage model. By substituting the current measurements in section Current Measurements on page 95, the overall average Icc for RDK keyboard can be calculated.

Table 8-2. RDK keyboard Power Consumption

	State	Hours/ day	Days	Average Icc (mA)	Charge (mAh)
Week day	Active	4	5	0.90	18
	Idle	20	5	0.041	4.1
Weekend	Idle	24	2	0.041	1.97
Charge Per Week (mAh)					24.07
Overall Average Icc (mA)					0.143

The RDK keyboard uses two AA cells and enables the PMU function. Therefore, it is able to access approximately 2850-mAh battery capacity, which yields a battery life estimate of 829 days or 27 months.

8.2 RDK Mouse

8.2.1 Usage Model

The following usage model is considered for the RDK mouse.

- 1 hour per day with the 3030/3040 sensor in ‘active’ state
- 2 hours per day with the 3030/3040 sensor in ‘rest1’ state
- 2 hours per day with the 3030/3040 sensor in ‘rest2’ state
- 19 hours per day with the 3030/3040 sensor in ‘rest3’ state
- 5 days per week as above, 2 days per week 24 hours in ‘rest3’ state

8.2.2 Current Measurements

The following is the results of RDK mouse current measurement:

Table 8-3. Mouse Current Measurement

Operation State	Icc (mA) with Supply Voltage = 2.5 V	Icc (mA) with Supply Voltage = 2.8 V	Average Icc (mA)
Active state - Move the mouse in a circle on white paper.	7.7	6.4	7.1
Rest1 state - Allow the mouse to sit idle for 1 second after being in the active state.	1.5	1.2	1.4
Rest2 state - Allow the mouse to sit idle for 10 seconds after being in the active state.	0.15	0.12	0.14
Rest3 state - Allow the mouse to sit idle for 10 minutes after being in the active state.	0.07	0.05	0.06
Not connected state - Move the mouse.	23.2	19.1	21.2
Not connected state - No moving.	Transition to rest3 state.	Transition to rest3 state.	

8.2.3 Battery Life Calculations

The following table shows the times spent in each state by the RDK mouse usage model. By substituting the current measurements in section Current Measurements on page 95, the overall average Icc for RDK mouse can be calculated.

Table 8-4. RDK Mouse Power Consumption

	State	Hrs/day	Days	Average Icc (mA)	Charge (mAh)
Week day	Active	1	5	7.05	35.25
	Rest1	2	5	1.37	13.7
	Rest2	2	5	0.15	1.5
	Rest3	19	5	0.06	5.7
Weekend	Rest3	24	2	0.06	2.88
Charge Per Week (mAh)					59.0
Overall Average Icc (mA)					0.351

The RDK mouse uses two AA cells and enables the PMU function. Therefore, it is able to access approximately 2850-mAh battery capacity, which yields a battery life estimate of 338 days or 11 months.

9. Software Users Guide

9.1 Introduction

This section describes the software source code modules used in order to communicate with the WirelessUSB LP bridge HID device to obtain the current radio parameters for the attached WirelessUSB devices. It does not cover the details of the Microsoft Foundation Class (MFC) Library or the HID Library that contains standard system-supplied routines that user-mode applications use to communicate with USB devices that comply with the USB HID Standard. Refer to the Microsoft Visual C++ documentation for more on MFC and HID Class concepts, in addition to the Device Class Definition for Human Interface Devices (HID) defined by the USB Implementers Forum, Inc. (http://www.usb.org/developers/hidpage).

9.2 Software Code Modules

There are three main modules contained in the WirelessUSB Software:

- USB HID API module – generic class interface to HID Class compliant devices
- System Tray module – generic class to create and control an icon on the system tray
- WirelessUSB System Tray Application module – main system tray application module

The following sections describe the software module contents.

9.2.1 USB HID API module

The USB HID API module defines two classes, CHidDevice and CHidManager. The CHidDevice class is the primary interface to a HID device, while the CHidManager class keeps track of the arrival and removal of HID devices, along with notification to the application of such events. The building blocks for the USB HID API module was derived from the HCLIENT sample code provided in the Windows DDK. This module was designed to provide a generic interface to any HID Class compliant device and is not expected to require any modification, however all source code is provided for reference.

9.2.1.1 CHidDevice Class Methods

Table 9-1. CHidDeviceClass Methods

Method	Type	Description
OpenHidDevice()	Public	This method sets appropriate access rights, attempts to open a handle to the HID device, obtains the top collection data, and makes a call to setup input, output, and feature data buffers.
CloseHidDevice()	Public	This method closes the HID device handle, un-registers the HID device notification, and frees pre-parsed data and data/report buffers.
RegisterHidDevice()	Public	This method registers the HID device handle for event notification
IsOpen()	Public	This method is used to report if a valid handle is open to the HID device.
IsOpenForRead()	Public	This method is used to report if the handle open to the HID device allows for read access.
IsOpenForWrite()	Public	This method is used to report if the handle open to the HID device allows for write access
IsOpenOverlapped()	Public	This method is used to report if the handle open to the HID device allows for overlapped I/O.
IsOpenExclusive()	Public	This method is used to report if the handle open to the HID device is setup for exclusive access.
GetHandle()	Public	This method returns the handle to the HID device.
Read()	Public	This method reads an input report from the HID device, performs a validity check, and unpacks the report data.
Write()	Public	This method is used for every report ID, pack a report buffer and write the report data to the HID device.
GetFeature()	Public	This method obtains the feature report from each report ID exposed by the HID device.
SetFeature()	Public	This method sends a feature report for each report ID exposed by the HID device.
UnpackReport()	Public	This method scans through the HID report and if it can, fills in any data in the structures.
PackReport()	Public	This method packages the HID report based on the data in the structures.
GetManufacturerString()	Public	This method obtains the USB manufacturer string from the HID device.
GetProductString()	Public	This method obtains the USB product string from the HID device.
GetSerialNumberString()	Public	This method obtains the USB serial number string from the HID device.
RegGetValue()	Public	This method attempts to get a registry value from the registry key where the device-specific configuration information for the HID device is stored.
RegSetValue()	Public	This method attempts to set a registry value in the registry key where the device-specific configuration information for the HID device is stored.
SetupHidDevice()	Protected	This method sets up HID Input, Output and Feature data buffers used to simplify communication with HID devices.
ValidateHidDevice()	Protected	This method simply returns TRUE, it is expected that this routine will be overridden by the application where the actual validation will be handled.

9.2.1.2 CHidManager Class Methods

Table 9-2. CHidManagerClass Methods

Method	Type	Description
Create()	Public	This method creates an invisible window and uses the returned window handle to register for HID device notification events, then it creates a list of existing HID devices that are maintained by the HID manager.
IsHidDevicePresent()	Public	This method attempts to open a handle to the HID device to determine if it is present (or not) and returns the result
RefreshHidDevices()	Public	This method validates that all HID devices in the list are still present, removes those from the list that are currently not present, scans the list of all existing HID devices present, and then attempts to add the existing HID devices to the list
GetDeviceCount()	Public	This method returns the number of connected devices
GetFirstHidDevice()	Public	This method returns a pointer to the first HID device in the list
GetNextHidDevice()	Public	This method returns a pointer to the next HID device in the list
GetCurrentHidDevice()	Public	This method returns a pointer to the current HID device in the list
GetHidDeviceWithPath()	Public	This method scans the current list of HID devices and returns a pointer to the HID device that matches the device path provided
GetHidDevice WithHandle()	Public	This method scans the current list of HID devices and returns a pointer to the HID device that matches the device handle provided
HidDeviceAlreadyExists()	Public	This method determines if the HID device already exists in the list
AddHidDevice()	Public	This method checks if the provided HID device already exists, and if not, adds the new HID device to the end of the list, increments the HID device counter, and call the HID callback function to indicate a new HID device was added
RemoveHidDevice()	Public	This method closes the outstanding handle to the HID device, calls HID callback function to indicate HID device is being removed, removes the HID device from the list, and deletes the HID device
RemoveAllHidDevices()	Public	This method scans though all HID devices in the list and removes them
CreateUniqueDeviceID()	Public	This method attempts to create and maintain a unique ID for the associated HID device
FreeUniqueDeviceID()	Public	This method frees the specified unique ID
NewHidDevice()	Protected	This method allocates memory for a new HID device structure
DeleteHidDevice()	Protected	This method deletes previously allocated memory for an existing HID device structure
RegisterHidNotification()	Protected	This method registers for notification of events for all HID devices and calls HID callback function to indicate registration was completed
HidDeviceArrival()	Protected	This method makes sure the HID device does not already exist in the list, and then creates a new HID device, opens a handle to the device, adds the new HID device to the list, and registers event notification for this new HID device
HidDeviceQuery Removal()	Protected	This method readies the HID device for removal by making sure the handle is closed
HidDeviceRemoval()	Protected	This method removes the HID device

9.2.2 System Tray Module

The System Tray module defines the CCySysTray class which provides the interface to the system tray for the application. This module is not expected to require any modification; however, all source code is provided for reference.

9.2.2.1 CCySysTray Class Methods

Table 9-3. CCySysTray Methods

Method	Type	Description
Create()	Public	This method creates an invisible window and sets up the system tray icon (if needed)
SetIcon()	Public	This method sets (or replaces) the icon displayed on the system tray
RemoveIcon()	Public	This method removes the icon from the system tray
SetToolTip()	Public	This method sets the tool tip to be displayed on the system tray
SetMenuItem()	Public	This method sets the default menu item executed when the icon is double-clicked on the system tray
IsHidden()	Public	This method is used to determine if the system tray icon is hidden
ShowBalloonTip()	Public	This method displays a balloon style tip message (only supported on W2K or higher)
OnTrayNotification()	Public	This method processes events that occur to the icon in the system tray
OnTaskbarCreated()	Protected	This method is called when the system tray is being restarted (for example, if Explorer crashes)
WindowProc()	Protected	This method overrides the default WindowProc to call OnTrayNotification for messages targeting the system tray icon or OnTaskbarCreated if the system tray is being restarted

9.2.3 WirelessUSB System Tray Application Module

The WirelessUSB System Tray module is the main system tray application. This module places the icon on the system tray bar, manages the HID devices, displays pop up messages, and controls the WirelessUSB Status Property Sheet. Additionally, via command-line parameters, this module can enable and disable the system tray application from running at startup.

9.2.3.1 CWirelessUSBTrayApp Class Methods

The CWirelessUSBTrayApp class performs application initialization and removal, in addition it parses command-line parameters used to enable or disable the system tray application from being run at startup.

Table 9-4. CWirelessUSBTrayApp Methods

Method	Type	Description
InitInstance()	Public	This method performs basic initialization and checks for any command-line parameters: if command-line parameters are found, it takes the appropriate action and ends the application; if no command-line parameters are found then it checks to make sure the application is not currently running and, if not, then proceeds to run the system tray application
ExitInstance()	Public	This method performs some standard cleanup before the application ends
RegisterAutoLoader()	Protected	This method registers the application (itself) to always be run at startup and optionally launches itself as well
UnregisterAudoLoader()	Protected	This method un-registers the application (itself) to prevent running at startup and optionally ends itself from running
AutoLoadExe()	Protected	This method launches the specified EXE application

9.2.3.2 CMainFrame Class Methods

The CMainFrame class is the Visual C++ generated file that is a derived frame-window class for the system tray application's main frame window. This class has been modified to also perform the timer based polling of the WirelessUSB LP bridge HID device to obtain the radio parameters and display any appropriate pop up messages. Additionally, this class also processes the command message to create the WirelessUSB Status Property Sheet.

Table 9-5. CMainFrame Methods

Method	Type	Description
OnCreate()	Public	This method is called when a new window is created for this frame; it sets up the HID Notification callback, device status property sheet, initializes the HID manager, creates the system tray icon, sets up the menu and tool tips, if any HID devices are present then displays the icon on the system tray, and makes a call to start the timer
HIDNotification()	Public	This method processes notifications of when an HID device is added or removed from the list; it adds or removes property pages to the wireless status page and adds or removes the icon from the system tray when the first or last HID device is added or removed
OnStartTimer()	Public	This method starts the timer based on the hard-coded poll timer (currently set at once every 5 seconds)
OnStopTimer()	Public	This method stops the timer
OnTimer()	Public	This method is the timer routine that is called when the timer expires, it loops through all the HID devices in the list and updates their status values and then restarts the timer; also it occasionally requests an update in the battery level, currently set at once every hour
OnDestroy()	Public	This method is called when the frame window is destroyed; it stops the timer, removes the property sheet (if displayed), and removes the icon from the system tray
OnAppWireless USBStatus()	Public	This method displays the wireless status page, if it is not already displayed

9.2.3.3 *CWirelessUSBStatusPropertyPage Class Methods*

The CWirelessUSBStatusPropertyPage class is the Visual C++ generated file that implements the WirelessUSB Device Status Property Page, a unique property page is created for each WirelessUSB device enumerated.

Table 9-6. CWirelessUSBStatusPropertyPage Methods

Method	Type	Description
OnInitDialog()	Public	This method initializes the wireless status page, reads the current value of the Disable Warning Message check box from the registry, and makes a call to start the timer
OnDestroy()	Public	This method removes the wireless status page and stops the timer
OnStartTimer()	Public	This method starts the timer for the wireless status page based on the hard-coded poll timer (currently set at once ever 500 ms)
OnStopTimer()	Public	This method stops the timer for the wireless status page
CommaStr()	Public	This method takes a numeric value and returns a CString representation of the number with commas added
OnTimer()	Public	This method updates the HID device values displayed on the status page and then restarts the timer; also it occasionally requests an update in the battery level, currently set at once every 5 seconds while the status page is displayed
OnBnClickedWireless USBDisableWarning Message()	Public	This method is called when the Disable Warning Messages check box is changed; base on the check box value it either disables or enables battery and signal strength warning messages for the specific HID device; the updated value is stored in the device-specific configuration information for the HID device

9.2.3.4 *CWirelessUSBStatusPropertySheet Class Methods*

The CWirelessUSBStatusPropertySheet class is the Visual C++ generated file that implements the WirelessUSB Status Property Sheet, which generates a unique WirelessUSB Device Status Property Page for each WirelessUSB device enumerated.

Table 9-7. CWirelessUSBStatusPropertySheet Methods

Method	Type	Description
OnInitDialog()	Public	This method initializes the wireless status property sheet and adds a property page for each HID device in the list
OnBnClickedClose()	Public	This method ends the dialog box if the user selects the Close button

9.2.3.5 *CHidTrayDevice Class Methods*

The CHidTrayDevice class is derived from the CHidDevice class and is the class used to interface with WirelessUSB devices.

Table 9-8. CHidTrayDevice Methods

Method	Type	Description
RequestNewUsageValues()	Public	This method sets up and issues a Set Feature request to the HID device, which now simply requests the wireless device to provide an update of its battery level the next time it communicates with the USB bridge
UpdateUsageValues()	Public	This method retrieves the latest usage values from the USB bridge, which includes wireless channel, wireless PN code, last reported battery level, and signal strength
UpdateDeviceInfo()	Public	This method makes a call to update the HID device usage values and displays a warning message (if enabled)
GetUsageIDValue()	Public	This method extracts the value of the provided Usage ID from the feature data
VerifyHidDevice()	Public	This method is called to verify that the HID device is one that should be added to the list; right now this is done by making sure the usage page reported is WIRELESSUSB_USAGEPAGE and the usage reported is either WIRELESSUSB_USAGE_KEYBOARD or WIRELESSUSB_USAGE_MOUSE

9.2.3.6 *CHidTrayManager Class Methods*

The CHidTrayManager class is derived from the CHidManager class and is used to manage WirelessUSB devices.

Table 9-9. CHidTrayManager Methods

Method	Type	Description
NewHidDevice()	Protected	This method creates a new HID device, initializes it, and adds it to the list of existing HID devices
DeleteHidDevice()	Protected	This method removes the HID device from the list and deletes the HID device

9.3 Development Environment

The following tools are required to build and develop the Wireless USB Software application.

- Microsoft Visual C++ .NET
- Windows Driver Development Kit (DDK)

A Microsoft Windows based PC is used for tool execution.

The Microsoft Visual C++ .NET solution file can be found at the following location:

.\WirelessUSBSysTray\WirelessUSBTray.sln

Appendix A. References

CY4636 Getting Started

PSoC Designer version 4.3 documentation

CY3631 Manufacturing Test Kit

Device Class Definition for Human Interface Devices (HID) (http://www.usb.org/developers/hidpage)

Avago ADNS-3040 Low Power Optical mouse Sensor Data Sheet

CYRF6936 WirelessUSB™ LP 2.4 GHz Radio SoC Data Sheet

Index

F

H

I

K

L

M

N

O

P

R

S

U

W