



Value through Innovation

Pulse WIFI Dongle

.

8018XXXX-001 Rev. 50

13 July 2026

ID TECH

10721 Walker Street, Cypress, CA 90630-4720

Tel: (714) 761-6368 Fax (714) 761-8880

www.idtechproducts.com

support@idtechproducts.com

Copyright © 2025 ID TECH. All rights reserved.

ID TECH
10721 Walker St.
Cypress, CA 90630

This document, as well as the software and hardware described in it, is furnished under license and may be used or copied online in accordance with the terms of such license. The content of this document is furnished for information use only, is subject to change without notice, and should not be construed as a commitment by ID TECH. Reasonable effort has been made to ensure the accuracy of information provided herein. However, ID TECH assumes no responsibility or liability for any unintentional errors or inaccuracies that may appear in this document.

Except as permitted by such license, no part of this publication may be reproduced or transmitted by electronic, mechanical, recording, or otherwise, or translated into any language form without the express written consent of ID TECH. ID TECH, Concierge, and ViVOPay are trademarks or registered trademarks of ID TECH.

Warranty Disclaimer: The services and hardware are provided "as is" and "as-available" and the use of the services and hardware is at its own risk. ID TECH does not make, and hereby disclaims, any and all other express or implied warranties, including, but not limited to, warranties of merchantability, fitness for a particular purpose, title, and any warranties arising from a course of dealing, usage, or trade practice. ID TECH does not warrant that the services or hardware will be uninterrupted, error-free, or completely secure.

FCC Regulatory Compliance

This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

The user manual for an intentional or unintentional radiator shall caution the user that changes or modifications not expressly approved by the party responsible for compliance could void the user's authority to operate the equipment.

Note: The grantee is not responsible for any changes or modifications not expressly approved by the party responsible for compliance. Such modifications could void the user's authority to operate the equipment.

Note: This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates and uses can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and the receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

FCC RF Radiation Exposure Statement

This equipment complies with FCC radiation exposure limits set forth for an uncontrolled environment. To comply with FCC RF Exposure compliance requirements, this grant applies only to Mobile Configurations. The antennas used for the transmitter must be installed to provide a separation distance of at least 20cm from all persons and must not be co-located or operating in conjunction with any other antenna or transmitter.

ISED Warning statements

This device complies with Canada's licence-exempt RSSs. Operation is subject to the following two conditions:

(1) This device may not cause interference; and (2) This device must accept any interference, including interference that may cause undesired operation of the device.

Le présent appareil est conforme aux CNR d'Industrie Canada applicables aux appareils radio exempts de licence.

L'exploitation est autorisée aux deux conditions suivantes : (1) l'appareil ne doit pas produire de brouillage, et (2) l'utilisateur de l'appareil doit accepter tout brouillage radioélectrique subi, même si le brouillage est susceptible d'encompromettre le fonctionnement.

This equipment complies with IC RSS-102 radiation exposure limits set forth for an uncontrolled environment. This equipment should be installed and operated with a minimum distance of 20cm between the radiator and any part of your body.

Pour se conformer aux exigences de conformité CNR 102 RF exposition, une distance de séparation d'au moins 20 cm doit être maintenue entre l'antenne de cet appareil et toutes les personnes.

This Class B digital apparatus complies with Canadian ICES-003. Cet appareil numérique de la classe B est conforme à la norme NMB-003 du Canada.

CE Caution**RF Exposure**

This equipment complies with CE radiation exposure limits set forth for an uncontrolled environment. This equipment should be installed and operated with minimum distance of 20cm between the radiator and your body.

EU DECLARATION OF CONFORMITY

Hereby, ID TECH declares that the radio equipment type Pulse wifi dongle is in compliance with Directive 2014/53/EU.

The full text of the EU declaration of conformity is available at the following Internet address: <https://idtechproducts.com/>

The product is so constructed that it can be operated in at least one Member State without infringing applicable requirements on the use of radio spectrum.

Notice for Operating Frequency and Output Power:

Feature	Pulse wifi dongle
WIFI	Wi-Fi 2.4GHz: 19.91 dBm (2400 MHz ~ 2483.5 MHz)
	Wi-Fi 5GHz (U-NII-1): 19.89 dBm (5150 MHz ~ 5250 MHz)
	Wi-Fi 5GHz (U-NII-2A): 19.70 dBm (5250 MHz ~ 5350 MHz)
	Wi-Fi 5GHz (U-NII-2C): 19.47 dBm (5470 MHz ~ 5725 MHz)
	Wi-Fi 5GHz (U-NII-3): 13.70 dBm (5725 MHz ~ 5850 MHz)

Cautions and Warnings

Caution: The Pulse wifi dongle should be mounted 1-2 feet away from other Pulse wifi dongles. Can be adjusted based on lane setup.



Warning: Avoid close proximity to radio transmitters which may reduce the capability of the reader.

Note: 5.2 GHz帯域は屋内での使用のみに制限されています
 (5.2 GHz高電力基地局または中継局と通信する場合を除く)
 電波法により5.3 GHz帯は屋内使用に限ります

Translation: 5.2 GHz band is restricted to indoor use only
 (Except when communicating with 5.2GHz high power base stations or relay stations)
 5.3 GHz band is restricted to indoor use due to the Radio Law

Revision History

Date	Rev	Changes	By
7/13/2026	50	Initial Draft	

Table of Contents

1. OVERVIEW.....	5
2. PULSE DONGLE INTERFACE CONNECTORS	6
2.1. Dongle (1 to 1)	6
2.2. Dongle (1 to 2)	7
3. PULSE DONGLE INSTALLATION	8
4. PULSE DONGLE FIRMWARE CONFIGURATION AND API	9
4.1. Pulse Function Diagrams.....	9
4.1.1. <i>Pulse Input</i>	9
4.1.2. <i>Pulse Output</i>	9
4.2. Firmware API.....	10
4.2.1. <i>Opening the Pulse Device</i>	10
4.2.2. <i>Polling for Input Events</i>	10
4.2.3. <i>Reading Pulse Input</i>	11
4.2.4. <i>Writing Pulse Output</i>	11
4.2.5. <i>Setting Lockout and Bonus States</i>	12
4.2.6. <i>Closing the Device</i>	12
4.2.7. <i>Pulse Driver Data Structures</i>	13
4.3. Example Code	15
4.3.1. <i>pulse.h (Pulse Header file)</i>	15
4.3.2. <i>Pulse Output example code</i>	17
4.3.3. <i>Pulse Input example code</i>	18
4.3.4. <i>Lockout example code</i>	19
4.3.5. <i>Bonus example code</i>	20

1. Overview

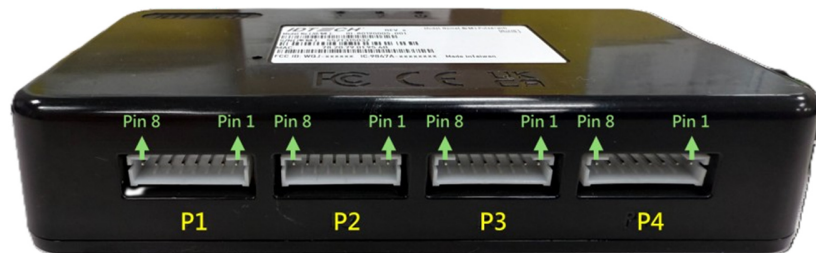
The VP7725 Pulse Dongle seamlessly modernizes existing coin-operated machines by enabling secure contactless card acceptance without requiring hardware modifications.



2. Pulse Dongle Interface Connectors

The following section provides details about the Pulse + Wi-Fi Dongle interface connectors and pin-out configurations.

2.1. Dongle (1 to 1)



Pin Definitions from P1 to P4

Pin	Name	Pin Type	Note
1	GND	GND	
2	OUTPUT1	Output	
3	OUTPUT2	Output	
4	INPUT1	Input	
5	INPUT2	Input	
6	LOCKOUT1	Output	
7	LOCKOUT2	Output	
8	BONUS	Output	

2.2. Dongle (1 to 2)

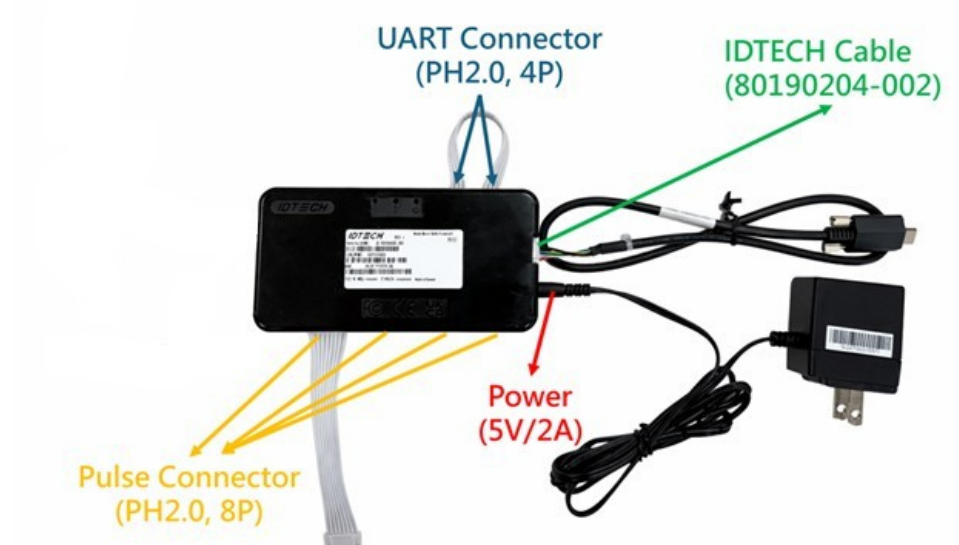


Pin Definitions from P5 to P6

P5 - Pin	Name	Pin Type	Note
1	VDD_3V3	Power	
2	RX	Output	
3	TX	Input	
4	GND	GND	
P6 - Pin	Name	Pin Type	Note
1	VDD_3V3	Power	
2	TX	Output	
3	RX	Input	
4	GND	GND	

3. Pulse Dongle Installation

The following section provides instructions for installing the Pulse Dongle.



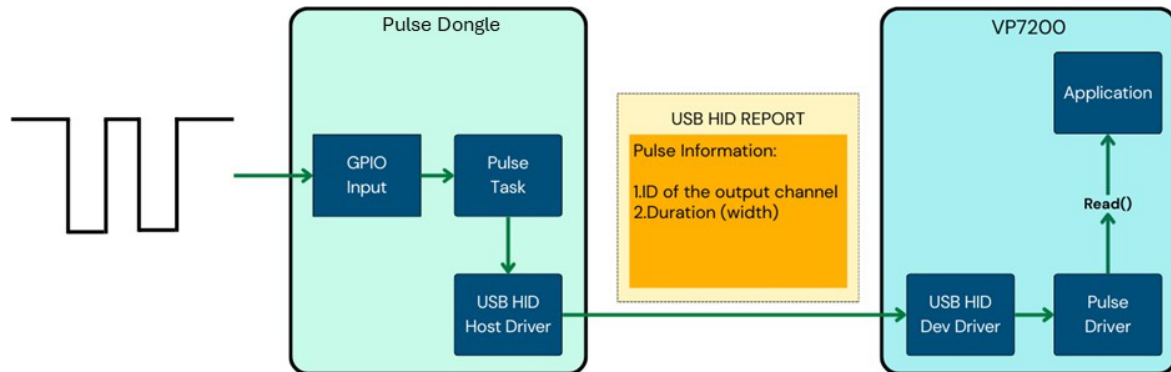
4. Pulse Dongle Firmware Configuration and API

The following section describes Pulse Dongle firmware configuration and API commands.

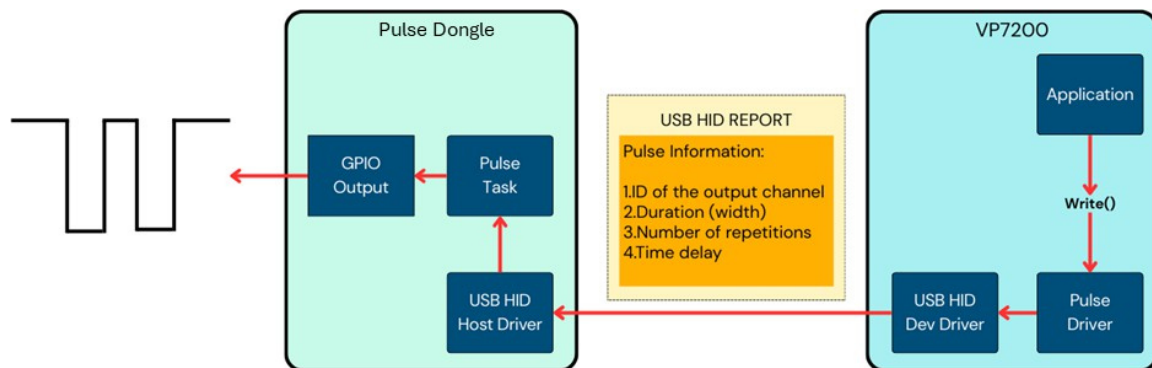
4.1. Pulse Function Diagrams

The following diagrams describe Pulse + Wi-Fi Dongle pulse input and pulse output.

4.1.1. Pulse Input



4.1.2. Pulse Output



4.2. Firmware API

The pulse driver is accessed and operated in NuttX using standard POSIX-like file I/O functions:

open()	Opens the pulse device.
poll()	Monitor pulse input availability
read()	Read input pulse data, including port id and input pin id.
write()	Write output pulse data to a specific port id and output id.
close()	Closes the pulse device.

4.2.1. Opening the Pulse Device

The **open()** function initializes access to the pulse driver. It must be opened with **O_RDWR** to support both pulse input monitoring and output command transmission.

Example Code

```

1.  /* Open the pulse driver */
2.  int fd = open("/dev/pulse", O_RDWR);
3.  if (fd < 0)
4.  {
5.      /* Failed to open device driver */
6.      return ERROR;
7.  }
8.

```

4.2.2. Polling for Input Events

The **poll()** function monitors the device for input activity across all ports. It allows the application to wait for an incoming pulse without consuming CPU resources. When a pulse is detected on any configured input pin, the **POLLIN** event is triggered.

Example Code

```

1.  struct pulse_io_s io;
2.  struct pollfd fds;
3.  /* Configure the polling structure */
4.  fds.fd = fd; // The file descriptor to monitor
5.  fds.events = POLLIN; // Wait for data to become available (Input Pulse)
6.  /* Block and wait for an event (timeout = -1 means infinite wait) */
7.  int ret = poll(&fds, 1, -1);
8.  /* Check if the poll was successful and the event was "Data Ready" */
9.  if (ret > 0 && (fds.revents & POLLIN))
10. {
11.     // Todo...
12. }
13.

```

4.2.3. Reading Pulse Input

After **poll()** indicates data is available, the **read()** function retrieves the specific pulse details. The driver populates the **pulse_io_s** structure with the source **Port** and measured **Width**.

Example Code

```

1.  struct pulse_io_s io;
2.
3.  /* Read the pulse data from the driver into the io structure */
4.  int read_len = read(fd, &io, sizeof(io));
5.  if (read_len == sizeof(io))
6.  {
7.      /* Display the captured pulse width only if it's an Input event */
8.      if (io.type == PULSE_INPUT_TYPE)
9.      {
10.         // Todo...
11.      }
12.  }
13.

```

4.2.4. Writing Pulse Output

The **write()** function sends commands to the driver to generate pulses. By setting **io.type** to **PULSE_OUTPUT_TYPE**, you can specify the target **Port**, **Pin**, and the exact waveform timing (Width, Count, and Delay).

Example Code

```

1.  /* Initialize the pulse configuration structure */
2.  struct pulse_io_s io;
3.
4.  /* Configure for Output Mode
5.   * Target specific hardware: Port 1, Pin 1
6.   * Define Pulse Waveform:
7.   * Width: 50ms active low
8.   * Count: 3 pulses in total
9.   * Delay: 200ms interval between pulses
10. */
11. io.type = PULSE_OUTPUT_TYPE;
12. io.output.port_id = PULSE_PORT_P1;
13. io.output.output_id = PULSE_OUTPUT_PIN1;
14. io.output.width = 50;
15. io.output.count = 3;
16. io.output.delay = 200;
17.
18. /* Send configuration to the driver to trigger the pulse generation */
19. int ret = write(fd, &io, sizeof(struct pulse_io_s));
20.

```

4.2.5. Setting Lockout and Bonus States

The **write()** function is also used to control auxiliary pins by switching the **io.type**.

- **Lockout:** Used to inhibit or allow signal acceptance on a specific port.
- **Bonus:** Used to trigger optional hardware features.

Example Code (Lockout)

```

1.  struct pulse_io_s io;
2.  int ret;
3.
4.  io.type = PULSE_LOCKOUT_TYPE;
5.  io.lockout.port_id = PULSE_PORT_P1;
6.  io.lockout.lockout_id = PULSE_LOCKOUT_PIN1;
7.  io.lockout.is_lock = 1; /* 1 = Lock (low active), 0 = Unlock */
8.
9.  /* Send configuration to the driver to set lockout */
10. ret = write(fd, &io, sizeof(struct pulse_io_s));
11.
```

Example Code (Bonus)

```

1.  struct pulse_io_s io;
2.  int ret;
3.
4.  io.type = PULSE_BONUS_TYPE;
5.  io.bonus.port_id = PULSE_PORT_P1;
6.  io.bonus.bonus_id = PULSE_BONUS_PIN1;
7.  io.bonus.is_bonus = 1; /* 1 = Active (Trigger Bonus) , 0 = Inactive */
8.
9.  /* Send configuration to the driver to set Bonus */
10. ret = write(fd, &io, sizeof(struct pulse_io_s));
11.
```

4.2.6. Closing the Device

The **close()** function closes the device and releases the file descriptor.

Example Code

```

1. close(fd);
```

4.2.7. Pulse Driver Data Structures

Pulse driver data structures define the formats used for reading input data from and writing output commands to the pulse device driver.

Structure	Direction	Purpose
struct pulse_input_s	Driver → App (Read)	Reports detected pulse inputs and pulse width.
struct pulse_output_s	App → Driver (Write)	Commands the driver to generate pulses.
struct pulse_lockout_s	App → Driver (Write)	Sets the states of the Lockout pins.
struct pulse_bonus_s	App → Driver (Write)	Sets the states of the Bonus pin.
struct pulse_io_s	Bidirectional	Main I/O wrapper structure.

4.2.7.1. struct pulse_input_s

This structure is used to return detected pulse information to the application when **read()** is called.

Definition:

```

1. /* Structure for Pulse Input events (Read from driver) */
2. struct pulse_input_s
3. {
4.     uint16_t port_id;    // The physical port (P1-P4)
5.     uint16_t input_id;   // The specific input pin (PIN1 or PIN2)
6.     uint16_t width;      // Measured pulse width duration (usually in ms)
7. };
8.

```

Fields:

Field	Type	Description
port_id	uint16_t	The physical hardware port identifier (e.g., PULSE_PORT_P1).
input_id	uint16_t	The specific input pin on the port (PIN1 or PIN2).
width	uint16_t	Width of the detected input pulse, in milliseconds.

4.2.7.2. struct pulse_output_s

The application uses this structure to request the driver to generate pulses via **write()**.

Definition:

```

1. /* Structure for Pulse Output commands (Write to driver) */
2. struct pulse_output_s
3. {
4.     uint16_t port_id;    // The physical port (P1-P4)
5.     uint16_t output_id;  // The specific output pin (PIN1 or PIN2)
6.     uint16_t width;      // Duration of the "Active" signal (ms)
7.     uint16_t count;      // Number of pulses to generate
8.     uint16_t delay;      // Interval between pulses (ms) if count > 1
9. };
10.

```

Description:

Field	Type	Description
port_id	uint16_t	Target hardware port for the pulse (P1-P4).
output_id	uint16_t	Target output pin on the port (PIN1 or PIN2).
width	uint16_t	Width of each output pulse, in milliseconds.
count	uint16_t	Total number of pulses to generate.
delay	uint16_t	Delay between consecutive pulses, in milliseconds.

4.2.7.3. struct pulse_lockout_s

The application uses this structure to request the driver to generate pulses via **write()**.

Definition:

```

1. /* Structure for Lockout/Inhibit control */
2. struct pulse_lockout_s
3. {
4.     uint16_t port_id;    // The physical port (P1-P4)
5.     uint16_t lockout_id; // The specific lockout pin (PIN1 or PIN2)
6.     uint16_t is_lock;    // State: 1 = Lock (Inhibit), 0 = Unlock (Allow)
7. };
8.

```

Description:

Field	Type	Description
port_id	uint16_t	Target hardware port for the pulse (P1-P4).
lockout_id	uint16_t	Target output pin on the port (PIN1 or PIN2).
is_lock	uint16_t	Sets the states of the Lockout pins

4.2.7.4. struct pulse_lockout_s

The application uses this structure to request the driver to generate pulses via **write()**.

Definition:

```

1. /* Structure for Bonus Pin control */
2. struct pulse_bonus_s
3. {
4.     uint16_t port_id;    // The physical port (P1-P4)
5.     uint16_t bonus_id;   // The specific bonus pin (PIN1)
6.     uint16_t is_bonus;   // State: 1 = Active, 0 = Inactive
7. };
8.

```

Description:

Field	Type	Description
port_id	uint16_t	Target hardware port for the pulse (P1-P4).
bonus_id	uint16_t	Target output pin on the port (PIN1 only).
is_bonus	uint16_t	Sets the states of the Bonus pin

4.3. Example Code

4.3.1. pulse.h (Pulse Header file)

```

1. #if !defined(__PULSE_H__)
2. #define __PULSE_H__
3.
4. #include <stdint.h>
5.
6. #ifdef __cplusplus
7. extern "C" {
8. #endif
9.
10. // -----
11. // SUB-STRUCTURES
12. // -----
13.
14. /* Structure for Pulse Input events (Read from driver) */
15. struct pulse_input_s
16. {
17.     uint16_t port_id;    // The physical port (P1-P4)
18.     uint16_t input_id;   // The specific input pin (PIN1 or PIN2)
19.     uint16_t width;      // Measured pulse width duration (usually in ms)
20. };
21.
22. /* Structure for Pulse Output commands (Write to driver) */
23. struct pulse_output_s
24. {
25.     uint16_t port_id;    // The physical port (P1-P4)
26.     uint16_t output_id;  // The specific output pin (PIN1 or PIN2)
27.     uint16_t width;      // Duration of the "Active" signal (ms)
28.     uint16_t count;      // Number of pulses to generate
29.     uint16_t delay;      // Interval between pulses (ms) if count > 1
30. };
31.
32. /* Structure for Lockout/Inhibit control */
33. struct pulse_lockout_s
34. {
35.     uint16_t port_id;    // The physical port (P1-P4)
36.     uint16_t lockout_id; // The specific lockout pin (PIN1 or PIN2)
37.     uint16_t is_lock;    // State: 1 = Lock (Inhibit), 0 = Unlock (Allow)
38. };
39.
40. /* Structure for Bonus Pin control */
41. struct pulse_bonus_s
42. {
43.     uint16_t port_id;    // The physical port (P1-P4)
44.     uint16_t bonus_id;   // The specific bonus pin (PIN1)
45.     uint16_t is_bonus;   // State: 1 = Active, 0 = Inactive

```

```

46. };
47.
48. // -----
49. // MAIN IO WRAPPER
50. // -----
51.
52. /* * This is the main structure passed to read()/write().
53.  * usage:
54.  * 1. Set 'type' to one of the PULSE_XX_TYPE macros.
55.  * 2. Fill in the corresponding structure inside the union.
56.  */
57. struct pulse_io_s
58. {
59.     uint16_t type; // Determines which struct inside the union is used
60.     union
61.     {
62.         struct pulse_output_s output;
63.         struct pulse_input_s input;
64.         struct pulse_lockout_s lockout;
65.         struct pulse_bonus_s bonus;
66.     };
67. };
68.
69. // -----
70. // CONSTANTS
71. // -----
72.
73. /* Command Types (Populate 'pulse_io_s.type' with these) */
74. #define PULSE_INPUT_TYPE    (0x00) // For reading detected pulses
75. #define PULSE_OUTPUT_TYPE   (0x01) // For generating pulses
76. #define PULSE_LOCKOUT_TYPE  (0x02) // For setting Inhibit/Lockout state
77. #define PULSE_BONUS_TYPE    (0x03) // For setting Bonus state
78.
79. /* Port Definitions (Total 4 Ports supported) */
80. #define PULSE_PORT_NUM      (0x04)
81.
82. #define PULSE_PORT_P1       (0x00)
83. #define PULSE_PORT_P2       (0x01)
84. #define PULSE_PORT_P3       (0x02)
85. #define PULSE_PORT_P4       (0x03)
86.
87. /* Pin Definitions per Port */
88. /* Output Pins (2 per port) */
89. #define PULSE_OUTPUT_PIN1    (0x00)
90. #define PULSE_OUTPUT_PIN2    (0x01)
91.
92. /* Input Pins (2 per port) */
93. #define PULSE_INPUT_PIN1     (0x00)
94. #define PULSE_INPUT_PIN2     (0x01)
95.
96. /* Lockout Pins (2 per port) */
97. #define PULSE_LOCKOUT_PIN1    (0x00)

```

```

98. #define PULSE_LOCKOUT_PIN2 (0x01)
99.
100. /* Bonus Pin (1 per port) */
101. #define PULSE_BONUS_PIN1 (0x00)
102.
103. #ifdef __cplusplus
104. }
105. #endif
106.
107. #endif /* __PULSE_H__ */
108.

```

4.3.2. Pulse Output example code

```

1. #include <nutt/config.h>
2. #include <nutt/pulse/pulse.h>
3. #include <nutt/fs/fs.h>
4. #include <fcntl.h>
5. #include <stdio.h>
6.
7. int main(int argc, FAR char *argv[])
8. {
9.     /* Open the pulse driver */
10.    int fd = open("/dev/pulse", O_RDWR);
11.
12.    if (fd < 0)
13.    {
14.        /* Failed to open device driver */
15.        return ERROR;
16.    }
17.
18.    /* Initialize the pulse configuration structure */
19.    struct pulse_io_s io;
20.
21.    /* Configure for Output Mode
22.     * Target specific hardware: Port 1, Pin 1
23.     * Define Pulse Waveform:
24.     * Width: 50ms active low
25.     * Count: 3 pulses in total
26.     * Delay: 200ms interval between pulses
27.     */
28.    io.type = PULSE_OUTPUT_TYPE;
29.    io.output.port_id = PULSE_PORT_P1;
30.    io.output.output_id = PULSE_OUTPUT_PIN1;
31.    io.output.width = 50;
32.    io.output.count = 3;
33.    io.output.delay = 200;
34.
35.    /* Send configuration to the driver to trigger the pulse generation */
36.    int ret = write(fd, &io, sizeof(struct pulse_io_s));

```

```

37.
38.     if (ret < 0)
39.     {
40.         /* Write operation failed */
41.         close(fd); // Best practice: close fd on error
42.         return ERROR;
43.     }
44.
45.     /* Close and Exit */
46.     close(fd);
47.
48.     return 0;
49. }
50.

```

4.3.3. Pulse Input example code

```

1. #include <nutt/config.h>
2. #include <nutt/pulse/pulse.h>
3. #include <nutt/fs/fs.h>
4. #include <fcntl.h>
5. #include <poll.h>
6. #include <stdio.h>
7.
8. int main(int argc, FAR char *argv[])
9. {
10.     /* Open the pulse driver */
11.     int fd = open("/dev/pulse", O_RDWR);
12.
13.     if (fd < 0)
14.     {
15.         /* Failed to open device driver */
16.         return ERROR;
17.     }
18.
19.     struct pulse_io_s io;
20.     struct pollfd fds;
21.
22.     /* Configure the polling structure */
23.     fds.fd = fd; // The file descriptor to monitor
24.     fds.events = POLLIN; // Wait for data to become available to read (Input
Pulse)
25.
26.     /* Block and wait for an event (timeout = -1 means infinite wait) */
27.     int ret = poll(&fds, 1, -1);
28.
29.     /* Check if the poll was successful and the event was "Data Ready" */
30.     if (ret > 0 && (fds.revents & POLLIN))
31.     {
32.         /* Read the pulse data from the driver into the io structure */
33.         int read_len = read(fd, &io, sizeof(io));

```

```

34.
35.     if (read_len == sizeof(io))
36.     {
37.         /* Display the captured pulse width only if it's an Input event */
38.         if (io.type == PULSE_INPUT_TYPE)
39.         {
40.             printf("Pulse detected! Width = %d ms\n", io.input.width);
41.         }
42.     }
43. }
44.
45. /* Close and Exit */
46. close(fd);
47.
48. return 0;
49. }
50.

```

4.3.4. Lockout example code

```

1. #include <nutt/config.h>
2. #include <nutt/pulse/pulse.h>
3. #include <nutt/fs/fs.h>
4. #include <fcntl.h>
5. #include <stdio.h>
6.
7. int main(int argc, FAR char *argv[])
8. {
9.     /* Open the pulse driver */
10.    int fd = open("/dev/pulse", O_RDWR);
11.
12.    if (fd < 0)
13.    {
14.        return ERROR;
15.    }
16.
17.    struct pulse_io_s io;
18.    int ret;
19.
20.    /* ===== */
21.    /* PHASE 1: ENGAGE LOCKOUT (Inhibit Acceptance) */
22.    /* ===== */
23.    printf("Engaging Lockout on P1...\n");
24.
25.    io.type = PULSE_LOCKOUT_TYPE;
26.    io.lockout.port_id = PULSE_PORT_P1;
27.    io.lockout.lockout_id = PULSE_LOCKOUT_PIN1;
28.    io.lockout.is_lock = 1; /* 1 = Lock (low active) */
29.
30.    ret = write(fd, &io, sizeof(struct pulse_io_s));
31.

```

```

32.  if (ret < 0)
33.  {
34.      close(fd);
35.      return ERROR;
36.  }
37.
38.  /* ===== */
39.  /* PHASE 2: HOLD (Wait 1 Second) */
40.  /* ===== */
41.  usleep(1000000);
42.
43.  /* ===== */
44.  /* PHASE 3: DISENGAGE LOCKOUT (Allow Acceptance) */
45.  /* ===== */
46.  printf("Disengaging Lockout on P1...\n");
47.
48.  io.type = PULSE_LOCKOUT_TYPE;
49.  io.lockout.port_id = PULSE_PORT_P1;
50.  io.lockout.lockout_id = PULSE_LOCKOUT_PIN1;
51.  io.lockout.is_lock = 0; /* 0 = Unlock */
52.
53.  ret = write(fd, &io, sizeof(struct pulse_io_s));
54.
55.  if (ret < 0)
56.  {
57.      close(fd);
58.      return ERROR;
59.  }
60.
61.  /* Close and Exit */
62.  close(fd);
63.
64.  return 0;
65. }
66.

```

4.3.5. Bonus example code

```

1.  #include <nutt/config.h>
2.  #include <nutt/pulse/pulse.h>
3.  #include <nutt/fs/fs.h>
4.  #include <fcntl.h>
5.  #include <stdio.h>
6.
7.  int main(int argc, FAR char *argv[])
8.  {
9.      /* Open the pulse driver */
10.     int fd = open("/dev/pulse", O_RDWR);
11.
12.     if (fd < 0)
13.     {

```

```

14.     return ERROR;
15. }
16.
17. struct pulse_io_s io;
18. int ret;
19.
20. /* ===== */
21. /* PHASE 1: ACTIVATE BONUS (Turn On) */
22. /* ===== */
23. printf("Activating Bonus Pin on P1...\n");
24.
25. io.type = PULSE_BONUS_TYPE;
26. io.bonus.port_id = PULSE_PORT_P1;
27. io.bonus.bonus_id = PULSE_BONUS_PIN1;
28. io.bonus.is_bonus = 1; /* 1 = Active (Trigger Bonus) */
29.
30. ret = write(fd, &io, sizeof(struct pulse_io_s));
31.
32. if (ret < 0)
33. {
34.     close(fd);
35.     return ERROR;
36. }
37.
38. /* ===== */
39. /* PHASE 2: HOLD (Wait 1 Second) */
40. /* ===== */
41. usleep(1000000);
42.
43. /* ===== */
44. /* PHASE 3: DEACTIVATE BONUS (Turn Off) */
45. /* ===== */
46. printf("Deactivating Bonus Pin on P1...\n");
47.
48. io.type = PULSE_BONUS_TYPE;
49. io.bonus.port_id = PULSE_PORT_P1;
50. io.bonus.bonus_id = PULSE_BONUS_PIN1;
51. io.bonus.is_bonus = 0; /* 0 = Inactive (Off) */
52.
53. ret = write(fd, &io, sizeof(struct pulse_io_s));
54.
55. if (ret < 0)
56. {
57.     close(fd);
58.     return ERROR;
59. }
60.
61. /* Close and Exit */
62. printf("Bonus test sequence complete.\n");
63. close(fd);
64.
65. return 0;

```

66. }
67.