

# Table of Contents

<b>Chapter 1</b>	<b>Introduction.....</b>	<b>3</b>
<b>Chapter 2</b>	<b>Validation Platform Setup .....</b>	<b>4</b>
2.1	AP33772 Sink Controller EVB .....	4
2.2	Arduino Uno .....	4
2.3	Validation Platform Connection and Power up.....	5
<b>Chapter 3</b>	<b>Arduino Software Setup .....</b>	<b>6</b>
3.1	Arduino IDE .....	6
3.1.1	Download the Arduino IDE (Windows) .....	6
3.1.2	Arduino IDE 1 Installation (Windows) .....	6
3.2	Setup of Arduino IDE .....	6
3.2.1	Board setting of Arduino IDE.....	6
3.2.2	Port setting of Arduino IDE.....	7
3.3	Run Arduino example .....	8
3.3.1	Upload sample code.....	8
3.3.2	Run Serial Monitor .....	8
3.3.3	Serial Monitor Baud Rate Configuration.....	9
3.3.4	The AP33772 I2C Tester.....	9
<b>Chapter 4</b>	<b>Basic Command Examples .....</b>	<b>11</b>
4.1	Arduino Wire Library.....	11
4.2	I2C read and write subroutine.....	11
4.2.1	i2c_read subroutine .....	11
4.2.2	i2c_write subroutine.....	11
4.3	I2C Command Examples.....	12
4.3.1	Read SRCPDO (0x00~0x1B).....	12
4.3.2	Read PDONUM (0x1C).....	12
4.3.3	Read STATUS (0x1D).....	12
4.3.4	Write MASK (0x1E) .....	12
4.3.5	Read VOLTAGE (0x20) .....	12
4.3.6	Read CURRENT (0x21).....	12
4.3.7	Read TEMP (0x22).....	12
4.3.8	Read and Write OCPTHR (0x23), OTPTHR (0x24), DRTHR (0x25).....	13
4.3.9	Read and Write TR25 (0x28~0x29), TR50 (0x2A~0x2B), TR75 (0x2C~0x2D), TR100 (0x2E~0x2F)..	13
4.3.10	Write RDO (0x30~0x33).....	14
4.3.11	Reset Command (0x30~0x33).....	14
<b>Chapter 5</b>	<b>Practical Examples.....</b>	<b>15</b>
5.1	Code Details .....	15
5.2	Code Execution and Outputs .....	19
5.3	Example Code Download.....	20
5.3.1	List of Example Code Files.....	20

5.3.2	Example Download Site.....	20
<b>Chapter 6</b>	<b>References.....</b>	<b>21</b>
<b>Chapter 7</b>	<b>Revision History.....</b>	<b>21</b>

## Chapter 1 Introduction

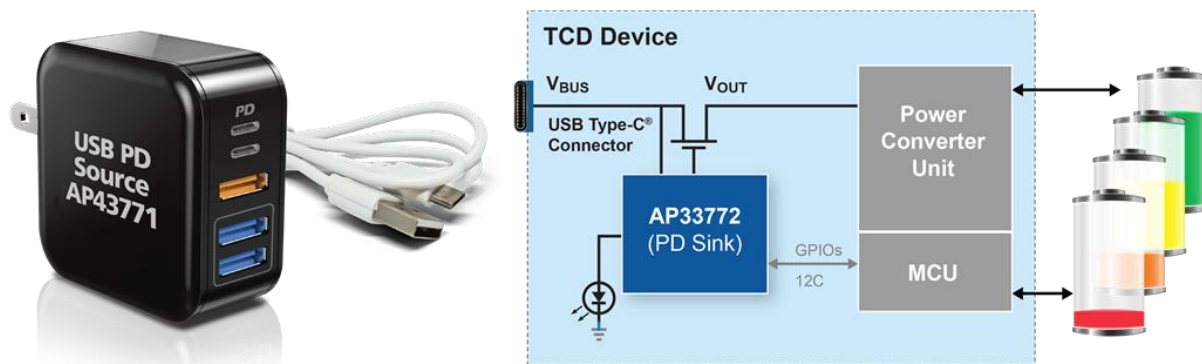
AP33772 Sink Controller, working as the protocol device of USB PD3.0 Type C Connector-equipped Device (**TCD**, Energy Sink), is intended to request proper Power Data Object (PDO) from the USB PD3.0 Type C Connector-equipped PD3.0 compliance Charger (**PDC**, Energy Source).

Figure 1 illustrates a TCD, embedded with PD3.0 Sink controller IC (AP33772), is physically connected to PDC, embedded with USB PD3.0 decoder (AP43771), through a Type C-to-Type C cable. Based on built-in USB PD3.0 compliant firmware, The AP33772 and AP43771 pair would go through the USB PD3.0 standard attachment procedure to establish suitable PD3.0 charging state.

AP33772 Sink Controller EVB provides ease of use and great versatility for system designer to request PDOs from USB Power Delivery Charger by sending AP33772 built-in commands through I2C interface. Typical system design requires MCU programming which needs specific software (e.g. IDE) setup and can be a time-consuming development process.

In contrast, Arduino, an open-source electronics platform based on easy-to-use hardware and software, are equipped with sets of digital and analog input/output (I/O) pins. Arduino provides a straightforward way to validate AP33772 Sink EVB working with a PD Charger. The goal of this guide is to provide system designers an effective platform to quickly complete software validation on Arduino and then port the development to any desirable MCU to meet rapid turnaround market requirements.

As a supplemental document to the AP33772 EVB User Guide, this User Guide illustrates an easy way to control AP33772 EVB with an Arduino through I2C Interface. The role of MCU block depicted in Figure 1 to interface with AP33772 is played by an Arduino. This User Guide covers a lot of register definition and usage information as examples, However, for complete and most updated information, please refer to AP33772 EVB User' Guide. (See Reference 2)



**Figure 1 – A typical TCD uses AP33772 PD Sink Controller with I2C Interface to request power from an USB Type-C PD3.0/PPS Compliance Source Adapter**

## Chapter 2 Validation Platform Setup

### 2.1 AP33772 Sink Controller EVB

Figure 2 shows the picture of the Sink Controller EVB. It features Type-C Connector, I2C pins, GPIO3 pin for Interrupt, NTC Thermistor for OTP, LED indicators to show the charging status, and Vout connector to the load.

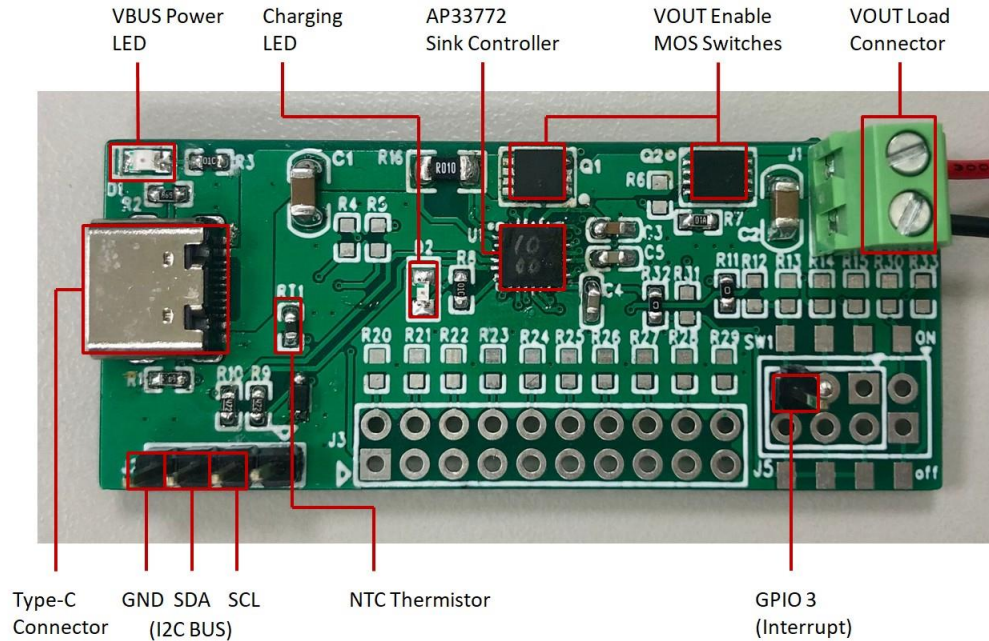


Figure 2 – AP33772 Sink Controller EVB

### 2.2 Arduino Uno

Any latest version of Arduino with I2C interface is capable of controlling AP33772 Sink Controller EVB. The Arduino Uno is the most used and documented board of the whole family. Simply connect it to a computer with a USB cable or power it with an AC-to-DC adapter or battery to get started. It serves the purpose as the AP33772 Sink Controller EVB Validation Platform perfectly.

User may check the Arduino official website for additional information. (<http://store.arduino.cc/products/arduino-uno-rev3>)

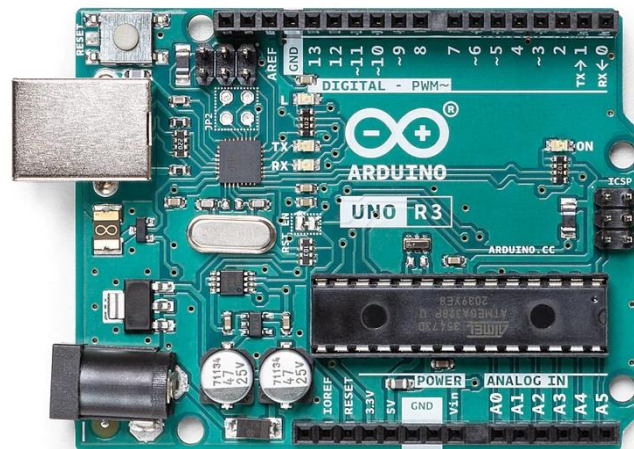


Figure 3 – Arduino Uno Rev3

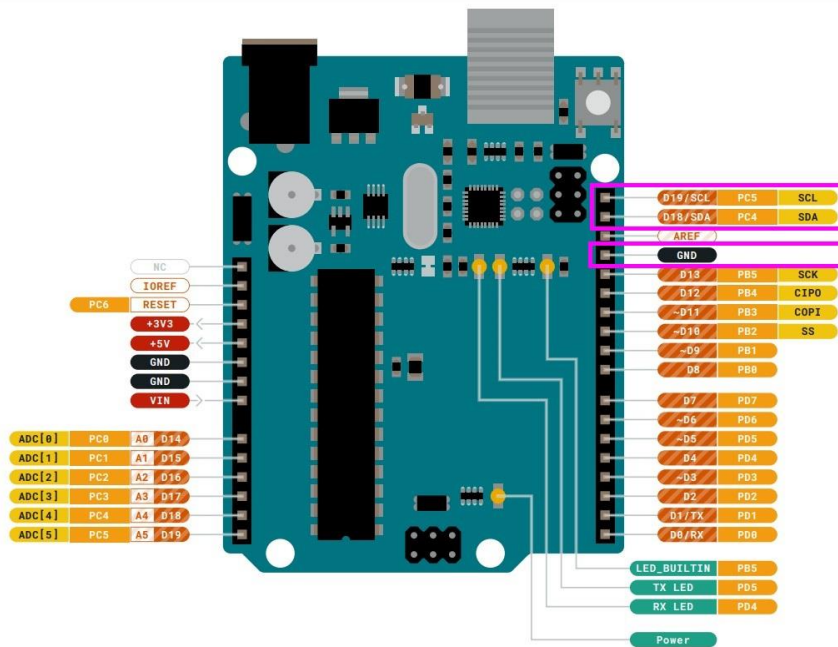


Figure 4 – Arduino Uno Rev3 Pinout Diagram

### 2.3 Validation Platform Connection and Power up

Figure 5 shows a complete connection and setup of the Validation Platform. User should follow these steps:

1. Connect SCL, SDA, and GND pins between Arduino and AP33772 EVB
2. Connect 65W PD Charger and AP33772 EVB with Type-C cable
3. Power up Arduino Uno
4. Power up PD Charger

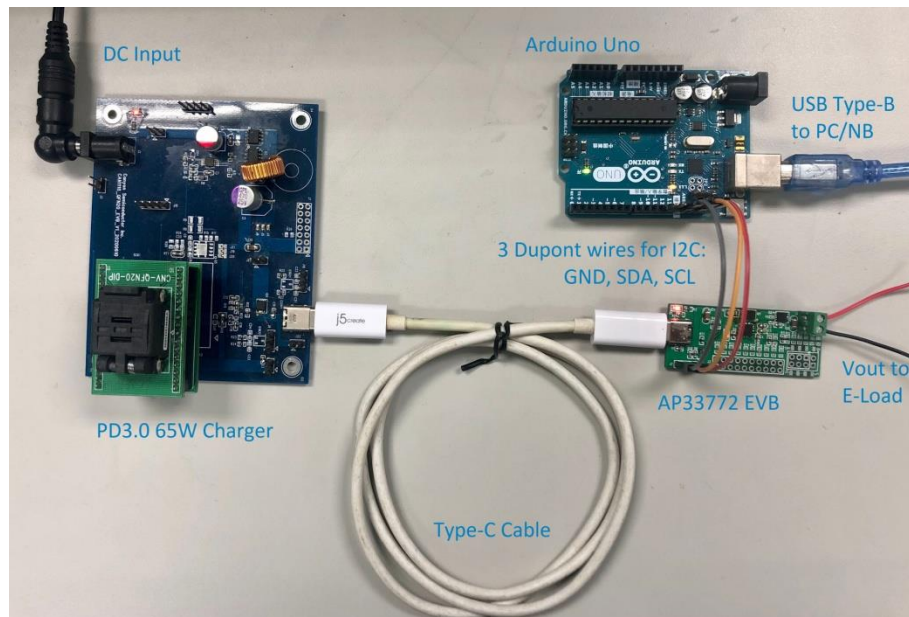


Figure 5 – Complete Setup of the Validation Platform

## Chapter 3 Arduino Software Setup

### 3.1 Arduino IDE

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board. Follow the instructions to get started with Arduino (<https://www.arduino.cc/en/Guide>). In the Arduino Software page you will find two options:

1. If you have a reliable Internet connection, you should use the online IDE (Arduino Web Editor).
2. If you would rather work offline, you should use the latest version of the desktop IDE.

The following examples in this user guide are based on the Windows 10 OS with Arduino desktop IDE.

#### 3.1.1 Download the Arduino IDE (Windows)

Get the latest version of Arduino Software (IDE) from the download page (<https://www.arduino.cc/en/software>) accordingly to your operating system.

## Downloads

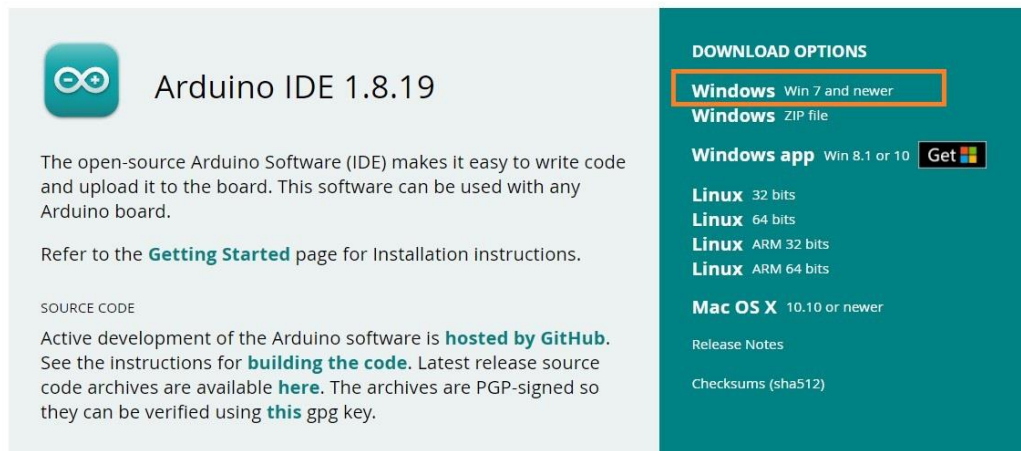


Figure 6 – Arduino IDE download options

#### 3.1.2 Arduino IDE 1 Installation (Windows)

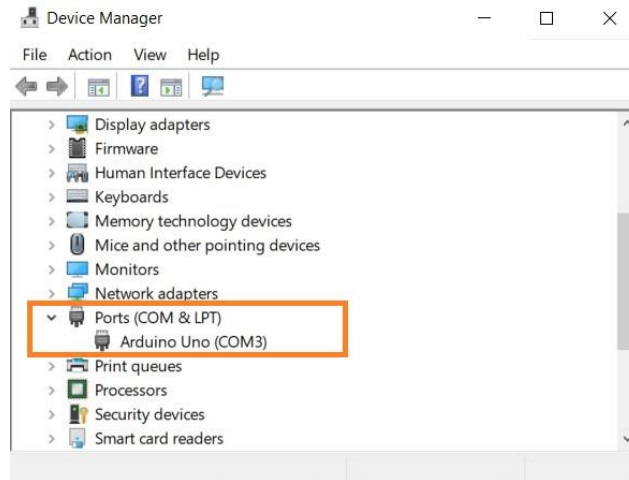
Follow the instructions to install the Arduino Software (IDE) on Windows machines (<http://docs.arduino.cc/software/ide-v1/tutorials/Windows>). Please allow the driver installation process when you get a warning from the operating system.

### 3.2 Setup of Arduino IDE

First time run the Arduino Software (IDE) after installation completed, please go to “Tools” and setup “Board” and “Port”.

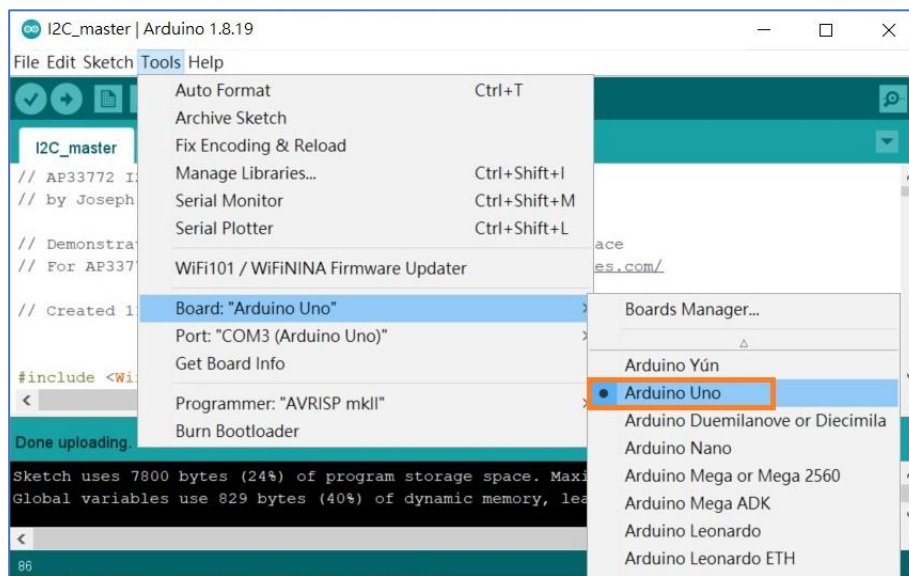
#### 3.2.1 Board setting of Arduino IDE

Connect the Arduino Uno to the PC/NB using a USB Type-B to Type-A cable. Then open the Device Manager and find out which COM port is occupied by the Arduino Uno. For example, the Arduino Uno occupies COM3 on the system in Figure 7.



**Figure 7 – Arduino Uno (COM3) under Ports (COM & LPT)**

Go to “Tools” and select “Arduino Uno” for Board setting.



**Figure 8 – Select Arduino Uno for Board setting**

### 3.2.2 Port setting of Arduino IDE

Then select the same COM number as the port in Device Manager for Prot setting. For example, go to “Tools” and select “COM3 (Arduino Uno)” for Port setting in Figure 9.

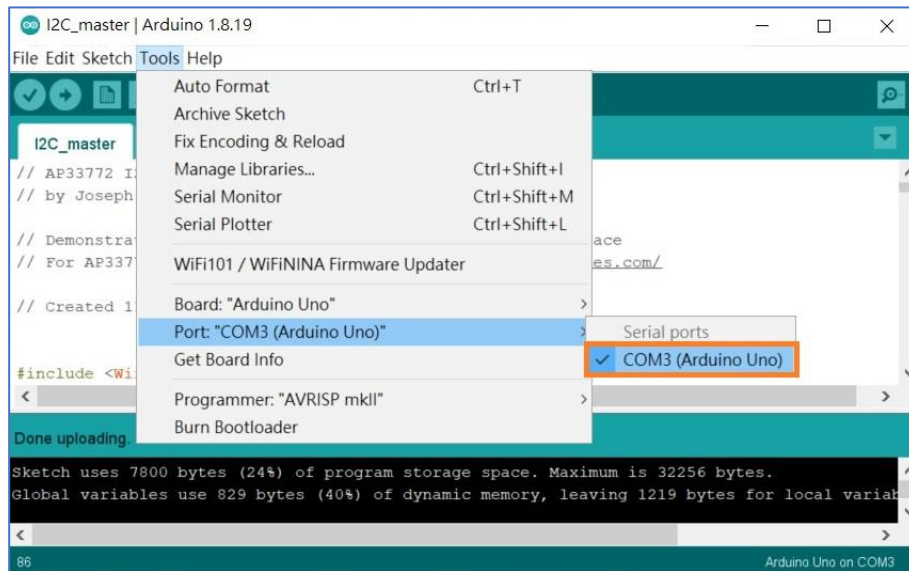


Figure 9 – Select COM3 (Arduino Uno) for Port setting

### 3.3 Run Arduino example

After the board and port are setup, the Arduino IDE can communicate with the Arduino Uno normally. Run I2C sample code “I2C\_master.ino” for AP33772 I2C evaluation.

#### 3.3.1 Upload sample code

Press “Upload” button to verify and upload the sample code to Arduino Uno. When the “Done uploading” message appears as shown in Figure 10, uploading the sample code is complete.

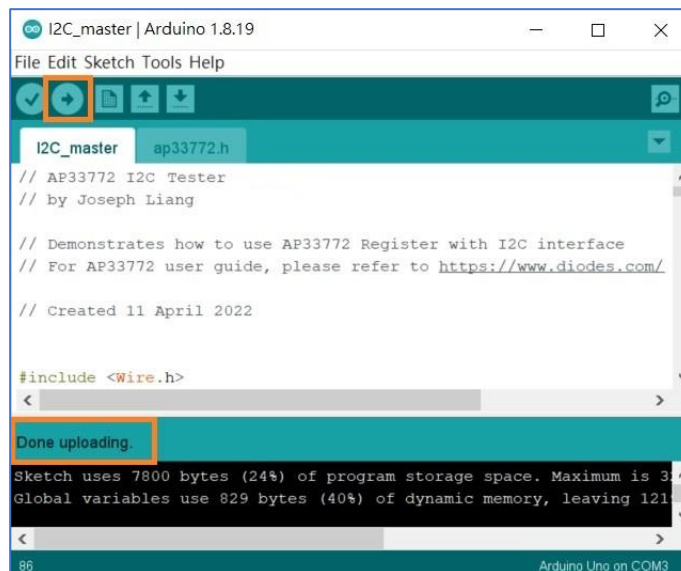
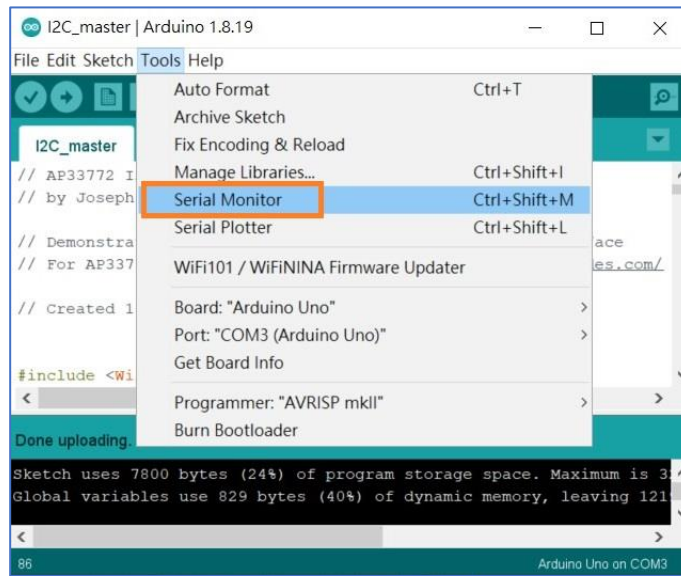


Figure 10 – Upload sample code

#### 3.3.2 Run Serial Monitor

Then go to “Tools” and run “Serial Monitor” as shown in Figure 11.

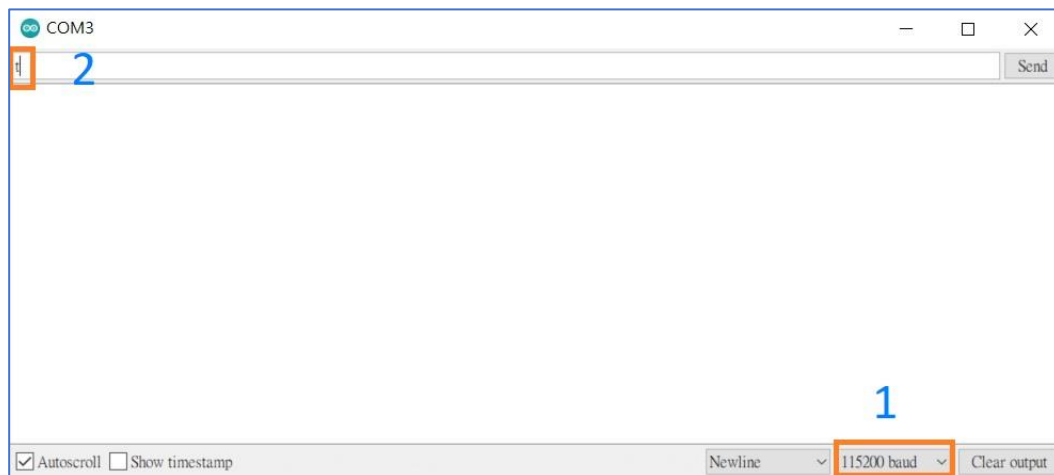




**Figure 11 – Run Serial Monitor**

### 3.3.3 Serial Monitor Baud Rate Configuration

When the Serial Monitor is running, first set the baud rate to 115200. Then enter “t” in the input block to start the AP33772 I2C Tester as shown in Figure 12.



**Figure 12 – Serial Monitor Baud Rate Configuration and Start Tester**

### 3.3.4 The AP33772 I2C Tester

When the “Starting Test ...” message appears in the output block, the AP33772 I2C Tester is running. Before connecting the PD source with the Type-C cable, make sure the SCL, SDA, and GND pins are connected between Arduino Uno and AP33772 EVB.

At the beginning, AP33772 I2C Tester will poll the status of AP33772 by reading the I2C register. After connecting the PD source and AP33772 EVB with Type-C cable, the tester gets the Source PDOs and displays it on the output block as shown in Figure 13.



**Figure 13 – Display the Source PDOs**

Enter the PDO position in the input block as requires by the user. The tester will construct and write the RDO register based on the selected PDO position. When the negotiation is completed and the PD source accepts the request, the “Voltage/Current” of VOUT and “Status” information will be displayed as shown in Figure 14.



**Figure 14 – Enter PDO Position and Information Display**

## Chapter 4 Basic Command Examples

This User Guide demonstrates how to use the I2C interface on an Arduino to communicate with the AP33772.

### 4.1 Arduino Wire Library

The Arduino Wire library allows user to communicate with I2C devices. This library provides beginTransmission, write, read, requestFrom, available and endTransmission commands. For complete information about the Wire library, please refer to <https://www.arduino.cc/reference/en/language/functions/communication/wire/>.

Table 1 shows the AP33772 register summary for user's convenience to digest the command usage in this section. For complete register information, please refer to AP33772 Sink Controller EVB User Guide.

Register	Command	Length	Attribute	Power-on	Description
SRCPDO	0x00	28	RO	All 00h	Power Data Object (PDO) used to expose PD Source (SRC) power capabilities. Total length is 28 bytes
PDONUM	0x1C	1	RO	00h	Valid source PDO number
STATUS	0x1D	1	RC	00h	AP33772 status
MASK	0x1E	1	RW	01h	Interrupt enable mask
VOLTAGE	0x20	1	RO	00h	LSB 80mV
CURRENT	0x21	1	RO	00h	LSB 24mA
TEMP	0x22	1	RO	19h	Temperature, Unit: °C
OCPTHR	0x23	1	RW	00h	OCP threshold, LSB 50mA
OTPTHR	0x24	1	RW	78h	OTP threshold, Unit: °C
DRTHR	0x25	1	RW	78h	De-rating threshold, Unit: °C
TR25	0x28	2	RW	2710h	Thermal Resistance @25°C, Unit: Ω
TR50	0x2A	2	RW	1041h	Thermal Resistance @50°C, Unit: Ω
TR75	0x2C	2	RW	0788h	Thermal Resistance @75°C, Unit: Ω
TR100	0x2E	2	RW	03CEh	Thermal Resistance @100°C, Unit: Ω
RDO	0x30	4	WO	0000000h	Request Data Object (RDO) is use to request power capabilities.
VID	0x34	2	RW	0000h	Vendor ID, Reserved for future applications
PID	0x36	2	RW	0000h	Product ID, Reserved for future applications
RESERVED	0x38	4	-	-	Reserved for future applications

Table 1 – AP33772 Register Summary

### 4.2 I2C read and write subroutine

The sample code “I2C\_master.ino” contains the “i2c\_read” and “i2c\_write” subroutines to simply read and write the AP33772 register. For more complete usage of subroutines, please refer to the source code.

#### 4.2.1 i2c\_read subroutine

The Arduino Wire.read command can only read one byte. The i2c\_read subroutine uses a while loop and saves the data in readBuf array.

```
Wire.beginTransmission(slvAddr);
Wire.write(cmdAddr);
Wire.endTransmission();

Wire.requestFrom(slvAddr, len);
while (Wire.available()) {
  readBuf[i] = Wire.read();
  i++;}
```

#### 4.2.2 i2c\_write subroutine

The Arduino Wire.write command has three syntaxes. Please write the data to writeBuf array first then call the i2c\_write subroutine.

```
Wire.beginTransmission(slvAddr);  
Wire.write(cmdAddr);  
Wire.write(writeBuf, len);  
Wire.endTransmission();
```

## 4.3 I2C Command Examples

Simplified usages are described in the examples under this section.

### 4.3.1 Read SRCPDO (0x00~0x1B)

The total length of Source PDO is 28 bytes. To read all PDO data, call the `i2c_read` subroutine as following. Then the PDO data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x00, 28);
```

### 4.3.2 Read PDONUM (0x1C)

To read the total number of valid PDOs, call the `i2c_read` subroutine as following. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x1C, 1);
```

### 4.3.3 Read STATUS (0x1D)

This command reports the Sink Controller's status including De-rating, OTP, OCP, OVP, NEWPDO, Negotiation Successful, and READY. To read the status information, call the `i2c_read` subroutine as following. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x1D, 1);
```

User should use this command after each RDO request to ensure a successful RDO request by reading the STATUS = 0x03.

### 4.3.4 Write MASK (0x1E)

This command enables the interrupts that signal the host through GPIO3 pin of AP33772. The interrupts include De-rating, OTP, OCP, OVP, NEWPDO, Negotiation Successful, and READY. To enable a specific interrupt, set the corresponding bit to one. For example, to enable OCP interrupt, set bit 5 of MASK register to one by call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x20;  
i2c_write(0x51, 0x1E, 1);
```

GPIO3 pin of AP33772 will go high when the OCP protection is trigger.

### 4.3.5 Read VOLTAGE (0x20)

This command reports the voltage measured by the AP33772 Sink Controller. To report the voltage, call the `i2c_read` subroutine as following. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x20, 1);
```

One unit of the reported value represents 80mV.

### 4.3.6 Read CURRENT (0x21)

This command reports the current measured by the AP33772 Sink Controller. To report the current, call the `i2c_read` subroutine as following. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x21, 1);
```

One unit of the reported value represents 24mA.

### 4.3.7 Read TEMP (0x22)

This command reports the temperature measured by the AP33772 Sink Controller. To report the temperature, call the `i2c_read` subroutine as following. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x22, 1);
```

One unit of the reported value represents 1°C.

#### 4.3.8 Read and Write OCPTHR (0x23), OTPTHR (0x24), DRTHR (0x25)

OCp, OTp, and De-rating thresholds can be changed to user desirable values by writing the values to OCPTHR, OTPTHR, and DRTHR registers. As an example, to change OCP threshold to 3.1A, user should write 0x3E (=3100/50=62=0x3E) to OCPTHR by call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x3E;  
i2c_write(0x51, 0x23, 1);
```

To change OTP threshold to 110°C, user should write 0x6E (=110) to OTPTHR by call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x6E;  
i2c_write(0x51, 0x24, 1);
```

To change De-rating threshold to 100°C, user should write 0x64 (=100) to DRTHR by call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x64;  
i2c_write(0x51, 0x25, 1);
```

To read the values out of OCPTHR, OTPTHR, and DRTHR, call the `i2c_read` subroutine as following respectively. The read data will be saved in the `readBuf` array.

```
i2c_read(0x51, 0x23, 1);  
i2c_read(0x51, 0x24, 1);  
i2c_read(0x51, 0x25, 1);
```

#### 4.3.9 Read and Write TR25 (0x28~0x29), TR50 (0x2A~0x2B), TR75 (0x2C~0x2D), TR100 (0x2E~0x2F)

A Murata 10KΩ Negative Temperature Coefficient (NTC) Thermistor NCP03XH103 is populated on the AP33772 EVB. It is user's preference to change the thermistor to a different one in the final design. User should update TR25, TR50, TR75, and TR100 register values according to specifications of the thermistor used. For example, Murata's 6.8KΩ NCP03XH682 is used in the design. The resistance values at 25°C, 50°C, 75°C, and 100°C are 6800Ω (0x1A90), 2774Ω (0x0AD6), 1287Ω (0x0507), and 662Ω (0x0296) respectively. To write the corresponding values to these registers by call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x90;  
writeBuf[1] = 0x1A;  
i2c_write(0x51, 0x28, 2);
```

```
writeBuf[0] = 0xD6;  
writeBuf[1] = 0x0A;  
i2c_write(0x51, 0x2A, 2);
```

```
writeBuf[0] = 0x07;  
writeBuf[1] = 0x05;  
i2c_write(0x51, 0x2C, 2);
```

```
writeBuf[0] = 0x96;  
writeBuf[1] = 0x02;  
i2c_write(0x51, 0x2E, 2);
```

To read the values out, call the `i2c_read` subroutine as following respectively. The read data will be saved in the `readBuf` array. The high byte is in `readBuf[1]` and the low byte is in `readBuf[0]`.

```
i2c_read(0x51, 0x28, 2);  
i2c_read(0x51, 0x2A, 2);  
i2c_read(0x51, 0x2C, 2);
```

```
i2c_read(0x51, 0x2E, 2);
```

#### 4.3.10 Write RDO (0x30~0x33)

To initiate a PDO request negotiation procedure, 4-byte data is written to RDO (Request Data Object) register in little-endian byte order. As example, to request PDO3 with 15V and 3A, 0x3004B12C will be written to RDO register. Call the `i2c_write` subroutine as following.

```
writeBuf[0] = 0x2C;  
writeBuf[1] = 0xB1;  
writeBuf[2] = 0x04;  
writeBuf[3] = 0x30;  
i2c_write(0x51, 0x30, 4);
```

Please refer to Table 10 and Table 11 of AP33772 Sink Controller EVB User Guide for detailed RDO content information.

#### 4.3.11 Reset Command (0x30~0x33)

User can issue a hard reset by writing RDO register with all-zero data. After issue the Reset Command, Arduino should disable the I2C module (`Wire.end`) and delay 1000ms then restart the I2C module (`Wire.begin`). This will ensure that the AP33772 Sink Controller can successfully complete the reset process.

```
writeBuf[0] = 0x00;  
writeBuf[1] = 0x00;  
writeBuf[2] = 0x00;  
writeBuf[3] = 0x00;  
i2c_write(0x51, 0x30, 4);  
  
Wire.end();  
delay(1000);  
Wire.begin();
```

The AP33772 Sink Controller will be reset to its initial state and output will be turned off.

## Chapter 5 Practical Examples

The AP33772 I2C Tester checks all valid PDOs and lists the voltage and current capability information out. The user can select the desired PDO by entering the PDO position. Please refer to Section 3.3.4 for usage.

### 5.1 Code Details

```
// AP33772 I2C Tester
// by Joseph Liang

// Demonstrates how to use AP33772 Register with I2C interface
// For AP33772 user guide, please refer to https://www.diodes.com/

// Created 11 April 2022

#include <Wire.h>
#include "ap33772.h"

#define SLAVE_ADDRESS    0x51
#define READ_BUFF_LENGTH  30
#define WRITE_BUFF_LENGTH  6
#define SRCPDO_LENGTH    28

byte readBuf[READ_BUFF_LENGTH] = {0};
byte writeBuf[WRITE_BUFF_LENGTH] = {0};
byte numPDO = 0;           // source PDO number
byte indexPDO = 0;        // PDO index, start from index 0
int reqPpsVolt = 0;       // requested PPS voltage, unit:20mV
bool startTesting = 0;

AP33772_STATUS_T ap33772_status = {0};
EVENT_FLAG_T event_flag = {0};
RDO_DATA_T rdoData = {0};
PDO_DATA_T pdoData[7] = {0};

void initWriteBuf()
{
  for(byte i=0 ; i<WRITE_BUFF_LENGTH ; i++)
  {
    writeBuf[i] = 0;
  }
}

void i2c_write(byte slvAddr, byte cmdAddr, byte len)
{
  Wire.beginTransmission(slvAddr); // transmit to device SLAVE_ADDRESS
  Wire.write(cmdAddr);             // sets the command register
  Wire.write(writeBuf, len);       // write data with len
  Wire.endTransmission();         // stop transmitting
  initWriteBuf();
}

void initReadBuf()
{
  for(byte i=0 ; i<READ_BUFF_LENGTH ; i++)
  {
    readBuf[i] = 0;
  }
}

void i2c_read(byte slvAddr, byte cmdAddr, byte len)
{
  byte i = 0;

  initReadBuf();
  Wire.beginTransmission(slvAddr); // transmit to device SLAVE_ADDRESS
  Wire.write(cmdAddr);             // sets the command register
  Wire.endTransmission();         // stop transmitting

  Wire.requestFrom(slvAddr, len); // request len bytes from peripheral device
  if (len <= Wire.available()) { // if len bytes were received
    while (Wire.available()) {
      readBuf[i] = (byte) Wire.read();
      i++;
    }
  }
}
}
```

```

void writeRDO()
{
  writeBuf[3] = rdoData.byte3;
  writeBuf[2] = rdoData.byte2;
  writeBuf[1] = rdoData.byte1;
  writeBuf[0] = rdoData.byte0;
  i2c_write(SLAVE_ADDRESS, CMD_RDO, 4); // CMD: Write RDO
}

void displayMainMenu()
{
  Serial.print("Select PDO [1~");
  Serial.print(numPDO);
  Serial.println("] or [9] Reset AP33772 :");
}

void checkStatus()
{
  i2c_read(SLAVE_ADDRESS, CMD_STATUS, 1); // CMD: Read Status
  ap33772_status.readStatus = readBuf[0];

  if(ap33772_status.isOvp)
    event_flag.ovp = 1;
  if(ap33772_status.isOcp)
    event_flag.ocp = 1;

  if(ap33772_status.isReady) // negotiation finished
  {
    if(ap33772_status.isNewpdo) // new PDO
    {
      if(ap33772_status.isSuccess) // negotiation success
        event_flag.newNegoSuccess = 1;
      else
        event_flag.newNegoFail = 1;
    }
    else
    {
      if(ap33772_status.isSuccess)
        event_flag.negoSuccess = 1;
      else
        event_flag.negoFail = 1;
    }
  }
  delay(10);
}

void eventProcess()
{
  if(event_flag.newNegoSuccess)
  {
    event_flag.newNegoSuccess = 0;
    Serial.println("=====");
    Serial.println("AP33772 I2C Tester");
    Serial.println();
    Serial.print("Read Status = 0x");
    Serial.print(ap33772_status.readStatus, HEX);
    Serial.println();

    i2c_read(SLAVE_ADDRESS, CMD_PDONUM, 1); // CMD: Read PDO Number
    numPDO = readBuf[0];
    Serial.print("Source PDO Number = ");
    Serial.print(numPDO);
    Serial.println();

    i2c_read(SLAVE_ADDRESS, CMD_SRCPDO, SRCPDO_LENGTH); // CMD: Read PDOs
    // copy PDOs to pdoData[]
    for(byte i=0; i<numPDO; i++)
    {
      pdoData[i].byte0 = readBuf[i*4];
      pdoData[i].byte1 = readBuf[i*4+1];
      pdoData[i].byte2 = readBuf[i*4+2];
      pdoData[i].byte3 = readBuf[i*4+3];
    }
  }
}

```



```

// display PDO information
Serial.println();
for(byte i=0 ; i<numPDO ; i++)
{
  if(((pdoData[i].byte3 & 0xF0) == 0xC0)    // PPS PDO
  {
    Serial.print("PDO[");
    Serial.print(i+1);    // PDO position start from 1
    Serial.print("] - PPS : ");
    Serial.print((float)(pdoData[i].pps.minVoltage) * 100 / 1000);
    Serial.print("V~");
    Serial.print((float)(pdoData[i].pps.maxVoltage) * 100 / 1000);
    Serial.print("V @ ");
    Serial.print((float)(pdoData[i].pps.maxCurrent) * 50 / 1000);
    Serial.println("A");
  }
  else if(((pdoData[i].byte3 & 0xC0) == 0x00) // Fixed PDO
  {
    Serial.print("PDO[");
    Serial.print(i+1);
    Serial.print("] - Fixed : ");
    Serial.print((float)(pdoData[i].fixed.voltage) * 50 / 1000);
    Serial.print("V @ ");
    Serial.print((float)(pdoData[i].fixed.maxCurrent) * 10 / 1000);
    Serial.println("A");
  }
}
Serial.println("=====");
displayMainMenu();
}

if(event_flag.negoSuccess)
{
  event_flag.negoSuccess = 0;
  delay(100);
  i2c_read(SLAVE_ADDRESS, CMD_VOLTAGE, 1); // CMD: Read VOLTAGE
  ap33772_status.readVolt = readBuf[0];
  i2c_read(SLAVE_ADDRESS, CMD_CURRENT, 1); // CMD: Read CURRENT
  ap33772_status.readCurr = readBuf[0];

  Serial.print(" >> Success : ");
  Serial.print(ap33772_status.readVolt*80);
  Serial.print("mV @ ");
  Serial.print(ap33772_status.readCurr*24);
  Serial.print("mA ; Status = 0x");
  Serial.println(ap33772_status.readStatus, HEX);
  Serial.println();
  displayMainMenu();
}

void setup() {
  Wire.begin(); // join i2c bus (address optional for master)
  Serial.begin(115200); // start serial for output
  Serial.setTimeout(10);
}

void loop() {
  byte cmd=0;
  byte input=0;

  if (Serial.available())
  {
    input = Serial.read();
  }

  if (input == 't') // Enter 't' to start Tester
  {
    Serial.println("Starting Test ...");
    startTesting = 1;
  }

  if(startTesting)
  {
    checkStatus();
    eventProcess();
  }
}

```

```

// Enter 1~9 valid
if ((input > '0') && (input <= '9'))
{
    input = input - '0';

    // Select PDO
    if ((input > 0) && (input <= numPDO))
    {
        indexPDO = input - 1; // Start from index 0
        if((pdoData[indexPDO].byte3 & 0xF0) == 0xC0) // PPS PDO
        {
            Serial.print("> Set PDO[");
            Serial.print(input);
            Serial.println("] - PPS PDO");

            // set request PPS voltage = max voltage
            reqPpsVolt = pdoData[indexPDO].pps.maxVoltage * 5; // convert unit: 100mV -> 20mV

            // set pps rdoData
            rdoData.pps.objPosition = input;
            rdoData.pps.opCurrent = pdoData[indexPDO].pps.maxCurrent;
            rdoData.pps.voltage = reqPpsVolt;
            writeRDO();
        }
        else if((pdoData[indexPDO].byte3 & 0xC0) == 0x00) // Fixed PDO
        {
            Serial.print("> Set PDO[");
            Serial.print(input);
            Serial.println("] - Fixed PDO");

            // set fixed rdoData
            rdoData.fixed.objPosition = input;
            rdoData.fixed.maxCurrent = pdoData[indexPDO].fixed.maxCurrent;
            rdoData.fixed.opCurrent = pdoData[indexPDO].fixed.maxCurrent;
            writeRDO();
        }
    }
}
else
{
    // Enter '9' for reset command test
    if(input == 9)
    {
        Serial.println("> Reset AP33772, please wait... ");
        Serial.println();
        writeBuf[0] = 0x00;
        writeBuf[1] = 0x00;
        writeBuf[2] = 0x00;
        writeBuf[3] = 0x00;
        i2c_write(0x51, 0x30, 4);
        // restart I2C module
        Wire.end();
        delay(1000);
        Wire.begin();
    }
    else
    {
        Serial.print("> Invalid PDO number : ");
        Serial.println(input);
        Serial.println();
        displayMainMenu();
    }
}
}
}
}

```

## 5.2 Code Execution and Outputs

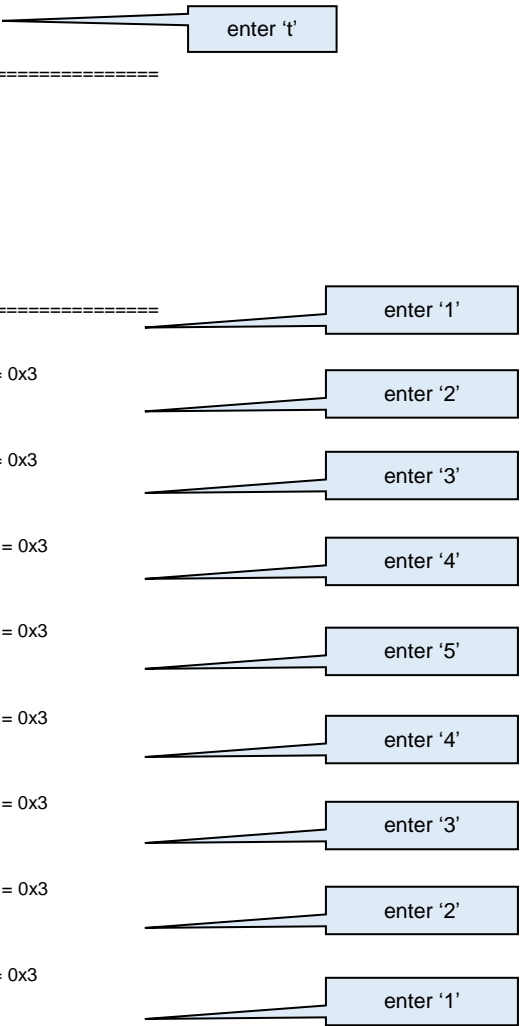
```

Starting Test ...
=====
AP33772 I2C Tester

Read Status = 0x7
Source PDO Number = 5

PDO[1] - Fixed : 5.00V @ 3.00A
PDO[2] - Fixed : 9.00V @ 3.00A
PDO[3] - Fixed : 15.00V @ 3.00A
PDO[4] - Fixed : 20.00V @ 3.00A
PDO[5] - PPS : 3.30V~21.00V @ 3.00A
=====
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[1] - Fixed PDO
>> Success : 5040mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[2] - Fixed PDO
>> Success : 8880mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[3] - Fixed PDO
>> Success : 14960mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[4] - Fixed PDO
>> Success : 20320mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[5] - PPS PDO
>> Success : 20400mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[4] - Fixed PDO
>> Success : 20320mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[3] - Fixed PDO
>> Success : 15040mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[2] - Fixed PDO
>> Success : 8880mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :
> Set PDO[1] - Fixed PDO
>> Success : 4960mV @ 96mA ; Status = 0x3
Select PDO [1~5] or [9] Reset AP33772 :

```



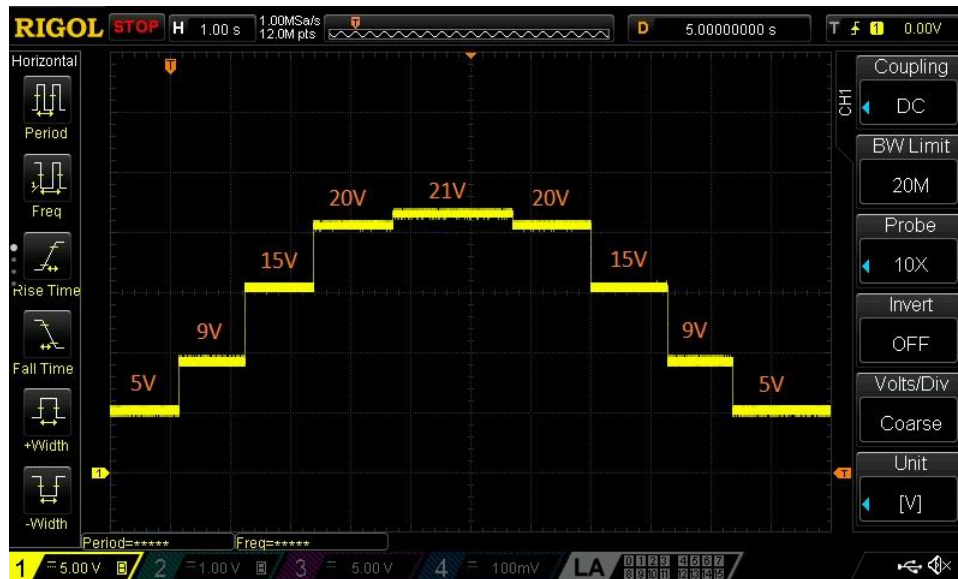


Figure 15 – Example of Output Waveform

## 5.3 Example Code Download

### 5.3.1 List of Example Code Files

There are two files in the AP33772 I2C Tester example code.

1. **I2C\_master.ino**: Main ino file of the AP33772 I2C Tester. Please use the Arduino IDE to open this file.
2. **ap33772.h**: The header file contains structure definitions. Make sure to be in the same folder as the ino file.

### 5.3.2 Example Download Site

Example Codes can be downloaded from Github with the following URL:

<https://github.com/diodinciot/ap33772-arduino>

## Chapter 6 References

1. AP33772 Datasheet (USB PD3.0 PPS Sink Controller): <https://www.diodes.com/products/power-management/ac-dc-converters/usb-pd-sink-controllers/>
2. AP33772 I2C Sink Controller EVB User Guide: <https://www.diodes.com/applications/ac-dc-chargers-and-adapters/usb-pd-sink-controller/>
3. Arduino Uno Rev3: <https://store.arduino.cc/products/arduino-uno-rev3>
4. Arduino Software: <https://www.arduino.cc/en/software>

## Chapter 7 Revision History

Revision	Issue Date	Comment	Author
1.0	4/15/2022	Initial Release	Joseph Liang

## IMPORTANT NOTICE

DIODES INCORPORATED MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARDS TO THIS DOCUMENT, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION).

Diodes Incorporated and its subsidiaries reserve the right to make modifications, enhancements, improvements, corrections or other changes without further notice to this document and any product described herein. Diodes Incorporated does not assume any liability arising out of the application or use of this document or any product described herein; neither does Diodes Incorporated convey any license under its patent or trademark rights, nor the rights of others. Any Customer or user of this document or products described herein in such applications shall assume all risks of such use and will agree to hold Diodes Incorporated and all the companies whose products are represented on Diodes Incorporated website, harmless against all damages.

Diodes Incorporated does not warrant or accept any liability whatsoever in respect of any products purchased through unauthorized sales channel.

Should Customers purchase or use Diodes Incorporated products for any unintended or unauthorized application, Customers shall indemnify and hold Diodes Incorporated and its representatives harmless against all claims, damages, expenses, and attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized application.

Products described herein may be covered by one or more United States, international or foreign patents pending. Product names and markings noted herein may also be covered by one or more United States, international or foreign trademarks.

This document is written in English but may be translated into multiple languages for reference. Only the English version of this document is the final and determinative format released by Diodes Incorporated.

## LIFE SUPPORT

Diodes Incorporated products are specifically not authorized for use as critical components in life support devices or systems without the express written approval of the Chief Executive Officer of Diodes Incorporated. As used herein:

A. Life support devices or systems are devices or systems which:

1. are intended to implant into the body, or
2. support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in significant injury to the user.

B. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or to affect its safety or effectiveness.

Customers represent that they have all necessary expertise in the safety and regulatory ramifications of their life support devices or systems, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of Diodes Incorporated products in such safety-critical, life support devices or systems, notwithstanding any devices- or systems-related information or support that may be provided by Diodes Incorporated. Further, Customers must fully indemnify Diodes Incorporated and its representatives against any damages arising out of the use of Diodes Incorporated products in such safety-critical, life support devices or systems.