This application note describes using the I2S peripheral of INP1010/INP1011/INP1012/INP1013 Talaria TWO modules for playing audio.

# I2S

I2S is one of the peripherals on Talaria TWO used for transmitting audio data. The I2S peripheral supports HD and dual channel stereo audio formats. A maximum sampling rate of 48kHz is supported. I2S can be designed to be configured on any of the Talaria TWO GPIOs except GPIO17. A separate I2S-to-Analog convertor along with Audio Amplifier must be used along with Talaria TWO. In the given example MAX98357A I2S-to-Analog convertor used,as indicated in Figure 1.

Diagram

Description automatically generated

Figure 1: Block Diagram

Application Reference Diagram: DAC used MAX98357A along with Talaria TWO I2S peripheral.

|  |  |
| --- | --- |
| **I2S Specification** | **Details** |
| Modes of operation | Master, Slave |
| Audio formats supports | Up to HD audio, Dual channel stereo |
| Supported sampling rates | 11025, 12000,22050, 24000,44100,48000 |

Table 1: I2S specification with details

# Required Dependencies on Ubuntu Host

Install the dependencies - python3, numpy, mpg123 on the ubuntu host machine:

|  |
| --- |
| sudo apt update  sudo apt install python3.8  sudo apt install python3-pip  pip install numpy |

# Source Code Walkthrough

## Directory Structure

Figure 2: Directory structure

1. **i2s\_audio\_flash.c -** The i2s\_audio\_flash.c file present in this directory contains the logic to configure the i2s and reads the audio file stored in file system and plays the audio over I2S.
2. **sound.c -** The sound.c file contains the sample audio data array.
3. **sound.h -** This header file contains structure definitions required to initialize the sample audio data buffer.
4. **i2s\_audio\_wifi**
   1. **audio\_server.c -** The audio\_server.c file present in this directory contains the logics to build and start a TCP/UDP server port, create an audio stream, receive the audio packets, and write the packets over I2S.
   2. **audio\_server.h** – The audio\_server.h header file contains the functions to build and start a TCP server port, create an audio stream, receive the audio packets, and write the packets over I2S.
   3. **i2s\_audio\_wifi.c -** The i2s\_audio\_wifi.c file contains logics to create Wi-Fi connectivity and starting an audio server.
   4. **tcp\_server.c -** The tcp\_server.c file contains the logics to set up a TCP server and sends and receives data over Wi-Fi after successfully connecting to the network.
   5. **tcp\_server.h** - The header file contains all the function prototypes need to create a TCP server.
   6. **udp\_server.c** - The udp\_server.c file contains the logic to set up a UDP server and sends and receives data over Wi-Fi after successfully connecting to the network.
   7. **udp\_server.h -** The header file contains all the function prototypes need to create a UDP server.

There are two examples to demonstrate the audio transmission over I2S of Talaria TWO:

1. i2s\_audio\_flash example that plays the audio stored in Talaria TWO’s file system.
2. i2s\_audio\_wifi example that receives the raw audio data over Wi-Fi and plays it over I2S.

## I2S APIs

* 1. Enabling I2S interface

The audio\_open() function Initializes an audio stream using the specified mode. I2S interface is set in this example.

* 1. Registering an audio call back function

The audio\_set\_callback()function registers a call back function that is invoked upon the completion of writing the packet containing the audio data.

* 1. Disabling I2S interface

The audio\_close() API releases the audio stream and shuts the audio device down. This invalidates the audio device.

* 1. Playing a file descriptor

The audio\_play() API reads the buffers the contents of the file descriptor and sends it to the DMA controller to play the audio.

* 1. Playing sample buffer

The audio\_play\_buffer() plays the sample audio data available in buffer.

## Code Walkthrough – I2s\_audio\_flash Example

**i2s\_audio\_flash.c**

The application sets the pin mux for the selected I2S pins based on the values of boot arguments. utils\_mount\_rootfs() function mounts the file system to be able to read the wav file from the file system.

audio\_open() function returns a pointer to the audio stream. audio\_play() API will play the audio from the .wav file stored in Talaria TWO’s filesystem and the audio\_play\_buffer() will play the audio from the sample audio buffer of the application.

The function os\_gpio\_mux\_sel() configures the GPIO to be used as the peripheral pin required for I2S functionality.

|  |
| --- |
| audio\_pins = BIT(audio\_gpio0) | BIT(audio\_gpio1) | BIT(audio\_gpio2);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_SCK, I2S\_CLK\_PIN);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_WS, I2S\_WS\_PIN);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_SD, I2S\_DATA\_PIN); |

If the file is available in specified location, then the fseek() seeks to end of file to get file size.

|  |
| --- |
| fseek(file, 0, SEEK\_END);  long file\_size = ftell(file);  fseek(file, 0, SEEK\_SET); |

After this, the pulse code modulation header and the audio data is extracted.

|  |
| --- |
| void\* file\_bytes = osal\_alloc(header\_size);  size\_t n = fread(file\_bytes, 1, header\_size, file);  pr\_debug("Read %d bytes of (%ld)\n", n, file\_size\_inc\_hdr);  wav = (struct wave\_file\*)file\_bytes; |

The data\_len is audio data length which is file length minus a potential audio header.

|  |
| --- |
| size\_t header\_size = sizeof(struct wave\_file);  data\_len = file\_size\_inc\_hdr - header\_size;  assert(file\_size\_inc\_hdr >= header\_size); |

The audio\_open() function initializes an audio stream using the specified mode, I2S in this example.

|  |
| --- |
| struct audio \*stream = audio\_open(AUDIO\_OUT\_I2S, playback\_rate, 0);  if(!stream) {  pr\_err("Failed to open audio stream.\n");  return 0;  } |

This loop starts the playback of audio signal through I2S. The audio\_play()reads and buffers the contents of the file descriptor and sends it in chunks directly to the DMA controller. The audio\_play\_buffer() plays the sample buffer. The buffer must contain samples of signed 16-bit values with the left and right channel interleaved.

|  |
| --- |
| pr\_info("Starting playback.\n");  int err = 0;  for(int i = 0; i < loops; i++) {  if (file) {  // Make sure the file is seeked to audio data position  fseek(file, sizeof(struct wave\_file), SEEK\_SET);  err = audio\_play(stream, file, data\_len);  } else {  err = audio\_play\_buffer(stream, sine\_wave\_size, sine\_wave);  }  if (err != 0) {  pr\_err("Failed to play audio.\n");  break;  }  } pr\_info("Playback finished.\n"); |

The audio\_close() releases the audio stream and shuts down the audio device by invalidating the struct audio pointer.

|  |
| --- |
| audio\_close(stream); |

## Evaluating i2s\_audio\_flash Example Application

The i2s\_audio\_flash example application reads the audio file stored in Talaria TWO’s file system and plays the audio over I2S. Hence, it is required to flash an audio file in .wav format to Talaria TWO’s filesystem before flashing the i2s\_audio\_flash.elf.

This example can also read the wave file from an audio buffer with a valid wave header and signed 16-bit data. Depending upon the value of a boot argument audio.playback\_mode (0: File, 1, static buffer), this example either reads the audio data from a wave file stored in Talaria TWO’s filesystem or from the audio buffer of the i2s\_audio\_flash application.

### Flashing Sample Wave File to Talaria TWO’s Filesystem

Program i2s\_audio\_flash.elf(*freertos\_sdk\_x.y\examples\i2s\_audio\bin*) using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the ELF by clicking on Select ELF File.
   3. Boot Arguments: Pass the following boot arguments:

|  |
| --- |
| audio.filename=/data/<file.wav>, audio.mode=1 |

* 1. File System:
     1. Select the \data\ folder from the path (*freertos\_sdk\_x.y\examples\i2s\_audio*) that contains the sample .wav file in Save Files to a Directory.
     2. Write this file to Talaria TWO.
  2. Programming: Click on Prog Flash.
  3. Show File System Contents: Click on Show File System Contents to check if the file is loaded. This writes the audio file to Flash.

**Note**:

* + 1. Audio file should be less than 900KB.

Console output:

|  |
| --- |
| UART:SNWWWWAE  4 DWT comparators, range 0x8000  Build $Id: git-8bc43d639 $  hio.baudrate=921600  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWAE  Build $Id: git-58974e3 $  Flash detected. flash.hw.uuid: 39483937-3207-0083-00a1-ffffffffffff  Bootargs: audio.filename=/data/plong.wav audio.mode=1  [0.018,670] Wav Samples: 22546  [0.018,699] Wav Frames: 11273  [0.018,729] Wav Chunk: 16  [0.018,757] Wav sample rate 44100  [0.018,792] Wav Duration: 0.5114  [0.019,099] Starting playback.  [0.528,699] Playback finished. |

Audio begins to play.

## Code Walkthrough – i2s\_audio\_wifi Example

The i2s\_audio\_wifi example works by receiving the audio data over Wi-Fi by starting a TCP or a UDP server. The client (PC in this case) sends the audio data over TCP/UDP socket.

In the following code walkthrough, the flow of creating a Wi-Fi interface, starting a TCP/UDP server, receiving the audio data packets, and writing the audio data to an I2S port is described.

**i2s\_audio\_wifi.c**

The main function starts with reading the boot argument values of SSID and passphrase that are required to connect with the Wi-Fi network. The wcm\_handle starts creating the Wi-Fi network interface. wcm\_notify\_enable() enables the callback function and IP address changes.

|  |
| --- |
| const char \*ssid = os\_get\_boot\_arg\_str("ssid") ?: "";  my\_wcm\_handle = wcm\_create(NULL);  wcm\_notify\_enable(my\_wcm\_handle, my\_wcm\_notify\_cb, NULL); |

network\_profile adds a network profile to Wi-Fi Connection Manager. The np\_conf\_path pointer variable contains the path to network configuration file on Talaria TWO’s file system. File and the path to the network configuration file is provided through the boot arguments.

|  |
| --- |
| /\*"/sys/nprofile.json"\*/  const char \*np\_conf\_path = os\_get\_boot\_arg\_str("np\_conf\_path")?: NULL;  struct network\_profile \*profile;  if (np\_conf\_path != NULL) {  /\* Create a Network Profile from a configuration file in  \*the file system\*/  rval = network\_profile\_new\_from\_file\_system(&profile, np\_conf\_path);  } else {  /\* Create a Network Profile using BOOT ARGS\*/  rval = network\_profile\_new\_from\_boot\_args(&profile);  }  if (rval < 0) {  pr\_err("could not create network profile %d\n", rval);  return 0; } |

The wcm\_add\_network\_profile() adds the network profile to WCM.

|  |
| --- |
| rval = wcm\_add\_network\_profile(my\_wcm\_handle, profile);  if (rval < 0) {  pr\_err("could not associate network profile to wcm %d\n", rval);  return 0; |

wcm\_auto\_connect() starts the auto connection with Wi-Fi network. os\_suspend\_enable() enables the device deep sleep mode via boot argument.

|  |
| --- |
| if(wcm\_auto\_connect(my\_wcm\_handle, 1) == 0)  if (os\_get\_boot\_arg\_int("suspend", 0) != 0)  os\_suspend\_enable(); |

os\_gpio\_mux\_sel() selects the GPIOs for I2S communication. The audio\_server() initiates the audio server on port 9999.

|  |
| --- |
| os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_SCK, I2S\_CLK\_PIN);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_WS, I2S\_WS\_PIN);  os\_gpio\_mux\_sel(GPIO\_MUX\_SEL\_I2S\_SD, I2S\_DATA\_PIN);  audio\_server(9999); |

struct wcm\_handle handles the Wi-Fi Connection Manager. This handle is an opaque representation of an interface managed by the Wi-Fi Connection Manager. The memory for this opaque struct is allocated in wcm\_create and freed in wcm\_destroy. The my\_wcm\_notify\_cb() is a Wi-Fi Connection Manager callback function.

|  |
| --- |
| struct wcm\_handle \*my\_wcm\_handle;  static void my\_wcm\_notify\_cb(void \*ctx, struct os\_msg \*msg)  os\_msg\_release(msg); |

**audio\_server.c**

This file contains the functions to build and start a TCP/UDP server port, create an audio stream, receive the audio packets, and write the packets over Wi-Fi.

The function server\_t\*build\_server() starts TCP/UDP server on initiated port according to the defined transport mode.

|  |
| --- |
| static inline server\_t\* build\_server(int port)  {  #ifdef TRANSPORT\_TCP  return tcp\_server(port);  #else  return udp\_server(port);  #endif  } |

The server\_accept() function allows the connection request from remote host i.e, the client. server\_rx() function initiates the TCP/UDP server data reception, by executing the tcp\_server\_rx()/udp\_server\_rx()functions.

|  |
| --- |
| static inline void server\_accept(server\_t \*srv)  {  #ifdef TRANSPORT\_TCP  tcp\_server\_accept(srv);  #endif  }  static inline int server\_rx(server\_t \*srv, void \*ptr, size\_t len)  {  #ifdef TRANSPORT\_TCP  os\_printf("tcp\_server\_rx\r\n");  return tcp\_server\_rx(srv, ptr, len);  #else  return udp\_server\_rx(srv, ptr, len);  #endif  } |

server\_cleanup() function frees the resources allocated, by executing the tcp\_ server\_cleanup ()/udp\_ server\_cleanup () functions.

|  |
| --- |
| static inline void server\_cleanup(server\_t \*srv)  {  #ifdef TRANSPORT\_TCP  return tcp\_server\_cleanup(srv);  #else  return udp\_server\_cleanup(srv);  #endif  } |

The function build\_server() binds the TCP/UDP server connection to a specific port, accepts the connections from the client by calling server\_accept().

|  |
| --- |
| server\_t \*server = build\_server(port); |

audio\_open() initializes an audio stream using the specified mode and the audio\_set\_callback()function registers a call back function that is invoked upon the completion of writing the packet containing audio data. If the audio is not enabled, it returns the start\_time, which is a current system time in microseconds.

|  |
| --- |
| #ifdef PWM\_AUDIO\_EN  struct audio \*stream = audio\_open(AUDIO\_OUT\_I2S, 48000, 0X1);  assert(stream);  audio\_set\_callback(stream, packet\_played);  num\_queued = 0;  #else  uint64\_t start\_time = os\_systime64();  uint64\_t bytes = 0;  #endif |

server\_rx() function receives the audio data packets from the client and inserts the packet to the packet linked list.

|  |
| --- |
| server\_rx(server, pfrag\_insert\_tail(frg, PACKET\_SIZE), PACKET\_SIZE) |

After receiving the audio data packets, audio\_write\_packet() writes the chunk of samples available in packet to the audio device over i2s.

|  |
| --- |
| audio\_write\_packet(stream, pkt);  num\_queued++;  os\_printf(".%d", num\_queued);  while (num\_queued>100); |

audio\_close() release the audio stream and shuts down the audio device. The server\_cleanup() cleans the server.

|  |
| --- |
| #ifdef PWM\_AUDIO\_EN  audio\_close(stream);  #endif  server\_cleanup(server); |

**tcp\_server.h**: This header file which contains all the function prototypes need to create a TCP server in Talaria TWO.

**tcp\_server.c:** This application sets up a tcp\_server and sends and receives data over Wi-Fi after successfully connecting to the network. TCP server accepts audio data using tcp\_server\_get\_bytes() function. It receives a data with a maximum value of 16-bit data and stores it into the buffer.

The audio\_server (int port) API receives raw data over TCP and plays the audio over I2S by writing the audio data using audio\_write\_packet() API.

The struct tcp\_server is declared with the all the parameter data needed to create a TCP server. The memory for this opaque struct is allocated in tcp\_server.

|  |
| --- |
| struct tcp\_server  {  struct netconn \*listen;  struct netconn \*conn;  struct netbuf \*buf;  void \*data;  uint16\_t len;  uint8\_t \*status;  }; |

The struct tcp\_server \* tcp\_server() creates the TCP server with the initialized port which will be the port used to created connection with clients.

|  |
| --- |
| os\_printf("Starting tcp-Server @ port %d\n", port);  struct tcp\_server \*server = osal\_zalloc(sizeof \*server);  assert(server); |

The netconn\_new() creates a new connection with the clients. netconn\_bind() binds the connection to a specific local IP address and port post which netconn\_listen()puts the TCP connection into listen state.

|  |
| --- |
| server->listen = netconn\_new(NETCONN\_TCP);  assert(server->listen != NULL);  netconn\_bind(server->listen, IP\_ADDR\_ANY, port);  netconn\_listen(server->listen);  return server; |

tcp\_server\_accept() function creates the TCP server and netconn\_accept() waits for a new incoming connection. This function blocks the process until a connection request from the remote host arrives.

|  |
| --- |
| void tcp\_server\_accept(struct tcp\_server \*srv)  {  netconn\_accept(srv->listen, &srv->conn);  os\_printf("TCP server: Accepted new connection %p\n", srv->conn);  } |

tcp\_server\_cleanup() function cleans up the TCP server and netconn\_delete() closes a net connection functions connection and frees the resources allocated .

|  |
| --- |
| void tcp\_server\_cleanup(struct tcp\_server \*srv)  { netconn\_delete(srv->conn);  } |

tcp\_server\_get\_bytes() function receives data over TCP, processes the data and stores it in the buffer.

|  |
| --- |
| static int tcp\_server\_get\_bytes(struct tcp\_server \*srv, void \*ptr, uint16\_t n) |

tcp\_server\_tx() function sends the TCP server data by calling. netconn\_write()to send data over a TCP connection.

|  |
| --- |
| int tcp\_server\_tx(struct tcp\_server \*srv, const void \*ptr, size\_t len)  {  netconn\_write(srv->conn, ptr, len, NETCONN\_COPY);  return 0;  } |

tcp\_server\_rx() function initiates the TCP server data reception by calling tcp\_server\_get\_bytes() function to receive the data over TCP, processes the data and store it in the buffer.

|  |
| --- |
| int tcp\_server\_rx(struct tcp\_server \*srv, void \*ptr, size\_t len)  { return tcp\_server\_get\_bytes(srv, ptr, len); } |

**udp\_server.h**: This header file contains all the function prototypes need to create a UDP server in Talaria TWO.

**udp\_server.c**:This application sets up a udp\_server and sends and receives data over Wi-Fi after successfully connecting to the network. UDP server accepts audio data using udp\_server\_get\_bytes()function. It receives data with a maximum value of 16-bit data and stores it in the buffer. audio\_server (int port) API receives raw data over UDP and plays the audio over I2S by writing the audio data using audio\_write\_packet() API.

The struct udp\_server is declared with the all the parameter data need to create a UDP server. The memory for this opaque struct is allocated in udp\_server.

|  |
| --- |
| struct udp\_server  {  struct netconn \*conn;  struct netbuf \*buf;  void \*data;  uint16\_t len;  uint8\_t \*status;  }; |

The struct udp\_server \*udp\_server() creates the UDP server with the initialized port which will be the port used to create connection with clients.

|  |
| --- |
| os\_printf("Starting udp-Server @ port %d\n", port);  struct udp\_server \*srv = osal\_zalloc(sizeof \*srv);  assert(srv); |

netconn\_new() creates a new connection with the clients through UDP. netconn\_bind() binds a connection to a specific local IP address and port.

|  |
| --- |
| srv->conn = netconn\_new(NETCONN\_UDP);  assert(srv->conn);  netconn\_bind(srv->conn, IP\_ADDR\_ANY, port);  return srv; |

The udp\_server\_cleanup() calls the netconn\_delete()API to close a connection and frees the resources allocated by calling osal\_free() .

|  |
| --- |
| void udp\_server\_cleanup(struct udp\_server \*srv)  {  if(srv->buf)  netbuf\_delete(srv->buf);  netconn\_delete(srv->conn);  osal\_free(srv);  } |

udp\_server\_get\_bytes() function receives the data over UDP, processes the data and stores it in the buffer.

|  |
| --- |
| static int udp\_server\_get\_bytes(struct udp\_server \*srv, void \*ptr, uint16\_t n) |

udp\_server\_rx() function initiates the UDP server data reception by calling the udp\_server\_get\_bytes() function to receive the data over UDP, processes it and store it in the buffer.

|  |
| --- |
| int udp\_server\_rx(struct udp\_server \*srv, void \*ptr, size\_t len)  {  return udp\_server\_get\_bytes(srv, ptr, len);  } |

## Evaluating the i2s\_audio\_wifi Example Application

The i2s\_audio\_wifi application contains by receiving the audio date over Wi-Fi by starting a TCP or a UDP server. The client (PC in this case) sends the audio data over TCP/UDP socket.

### Flashing a file to Talaria TWO’s Filesystem

Program i2s\_audio\_wifi.elf *(freertos\_sdk\_x.y\examples\i2s\_audio\bin)* using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the ELF by clicking on Select ELF File.
   3. AP Options: Provide the SSID and Passphrase under AP Options to connect to an Access Point.
   4. Programming: Click on Prog Flash.

Run the python script from the host PC to stream the audio raw data:

|  |
| --- |
| ./script/audio\_client.py <T2’s IP address> sample\_audio/Happy\_Birthday\_song\_50k.mp3 |

Console output:

|  |
| --- |
| 4 DWT comparators, range 0x8000  Build $Id: git-8bc43d639 $  hio.baudrate=921600  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWWWWAE  Build $Id: git-58974e3 $  Flash detected. flash.hw.uuid: 39483937-3207-0083-00a1-ffffffffffff  Bootargs: np\_conf\_path=/data/nprofile.json ssid=InnoIOT passphrase=InnoChip2023  addr e0:69:3a:00:15:b0  Connecting to added network : InnoIOT  Starting WiFi-Com-Server @ port 9999  Starting udp-Server @ port 9999  [0.939,297] CONNECT:60:22:32:60:06:52 Channel:1 rssi:-70 dBm  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_UP  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_ADDRESS  [3.676,131] MYIP 172.16.16.120  [3.676,180] IPv6 [fe80::e269:3aff:fe00:15b0]-link  .1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23.24.25.26.27.28.29.30.31.32.33.34.35.36.37.38.39.40.40.41.42.43.44.45.46.47.48.49.50.51.52.53.54.55.56.57.58.59.60.61.62.63.64.65.66.67.68.69.70.71.72.73.74.75.76.77.78.79.80.81.82.83.84.85.86.87.88.89. |