

TAS1020 Application Notes

Print Date: 3 October, 2000

Contact Information

Texas Instruments Incorporated

This specification is provided with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification or sample. No license, expressed or implied, by estoppel or otherwise, to any intellectual property rights is granted herein. Texas Instruments disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this document. Texas Instruments does not warrant or represent that such implementations will not infringe such rights.

1 Introduction

Writing normal application and the application based on the TAS 1020 ROM code are similar in general structures. The difference is that many generic functional modules in the normal application are provided by the TAS 1020 ROM code. These functional modules include:

- The USB Engine module which handles the USB data transaction for control endpoint 0.
- The Standard USB module which handles the Standard USB Chapter 9.
- The USB HID module which handles the USB HID Class.
- The USB DFU, Device Firmware Update, module which handles the USB DFU Class.
- The USB Audio module which handles the common features of the USB Audio Class.

This document describes methods and rules for writing the application to use the modules provided by the TAS 1020 ROM code. The demos are taken from the sample application compiled using Kyle compiler.

2 References

- USB specs. Version 1.0.
- USB Audio Class specs. Version 1.0.
- USB HID Class spec. version 1.0.
- USB DFU Class specs. Version 1.0.
- TUSB 1020 Data Manual.
- TAS 1020 Application Source Code.
- TAS 1020 Rom Source Code.

3 Conventions

- Application: The actual code written for applications using supports provided by the TAS 1020 ROM code.
- Normal application: The actual code written for application without using any support from the TAS 1020 ROM code.
- Rom code: the code stored in the ROM part of the TAS 1020 Chipset.

4 Application Models

This section presents the normal application and the TAS 1020 application for comparison. This will hopefully help us more in writing the TAS 1020 application.

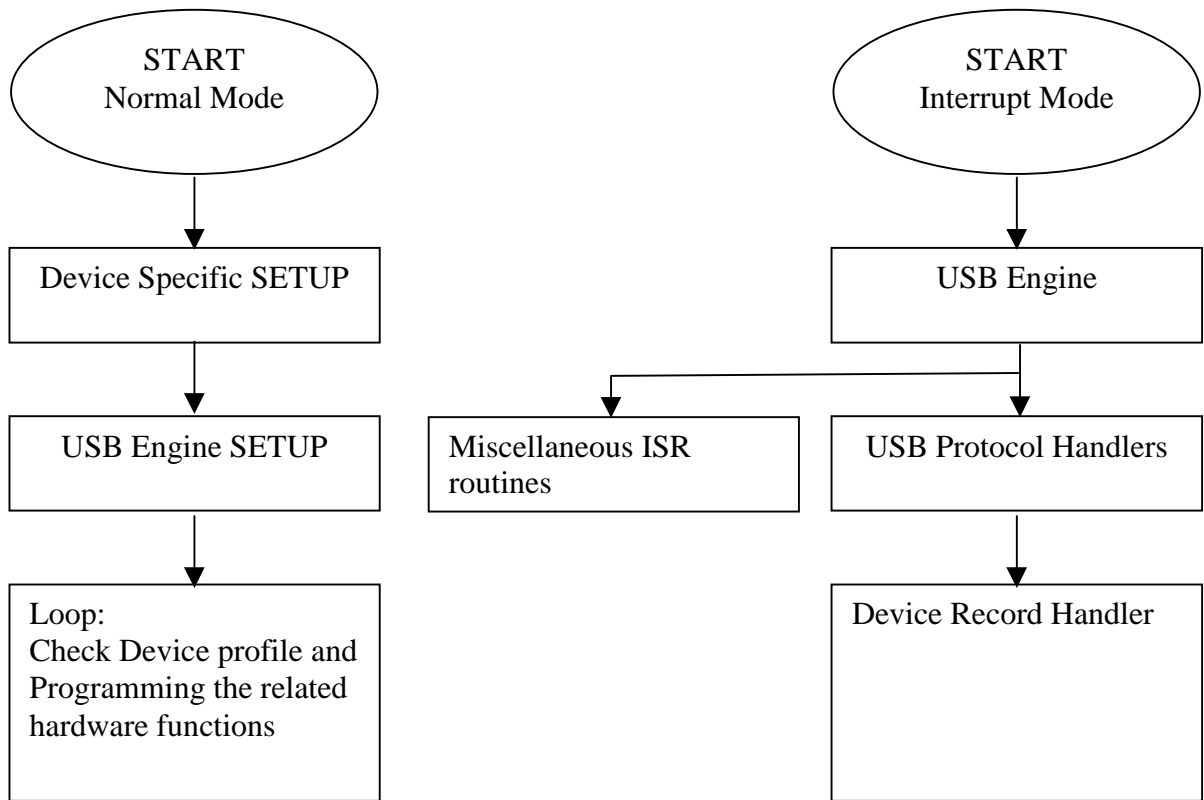
4.1 Normal Application

In a normal application which does not use the TAS 1020 ROM code, the application has to handle from A to Z. The main tasks are:

- USB Engine handling the USB Control Endpoint data transaction and states.
- USB Protocol Handlers for USB Chapter 9 Test, USB HID Class, USB DFU Class, and common features of USB Audio Class.
- Updating the device profile and programming the related hardware such as Codecs, GPIO pins, ...

The block diagram is presented in figure ...

Figure xxxx



4.2 TAS 1020 Application

The model for the TAS 1020 application is generally the same as the normal application. The only difference is that now many functional blocks are handled by the TAS 1020 ROM code. Main tasks handled by the TAS 1020 ROM code:

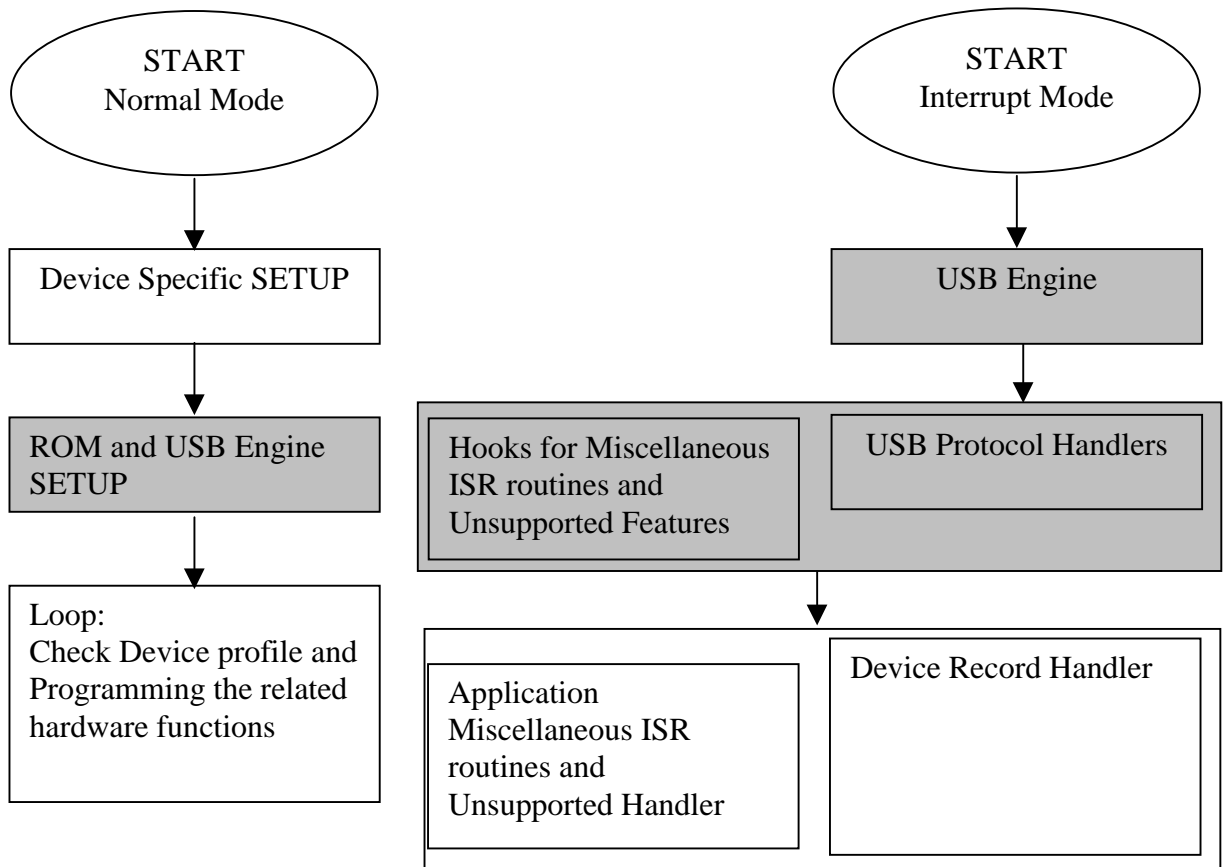
- USB Engine handling the USB Control Endpoint data transaction and states.
- USB Protocol Handlers for USB Chapter 9 Test, USB HID Class, USB DFU Class, and common features of USB Audio Class.

Main tasks handled by the specific application code:

- Updating the device profile and programming the related hardware such Codec, GPIO pins, ...
- Device Record and Related Hardware Handler: Handling the device profile and miscellaneous interrupt service routines.

The block diagram is presented in figure ... The shading part contains functional block provided by the TAS 1020 ROM Code.

Figure xxx



4.3 TAS 1020 Application Overview

From Figure ..., the TAS 1020 application handles the following main tasks in the normal mode:

- Device Specific Setup.
- Setup the ROM/USB engine/DFU if applied by calling the TAS 1020 ROM functions.
- Looping on checking device profile and programming the related hardware.

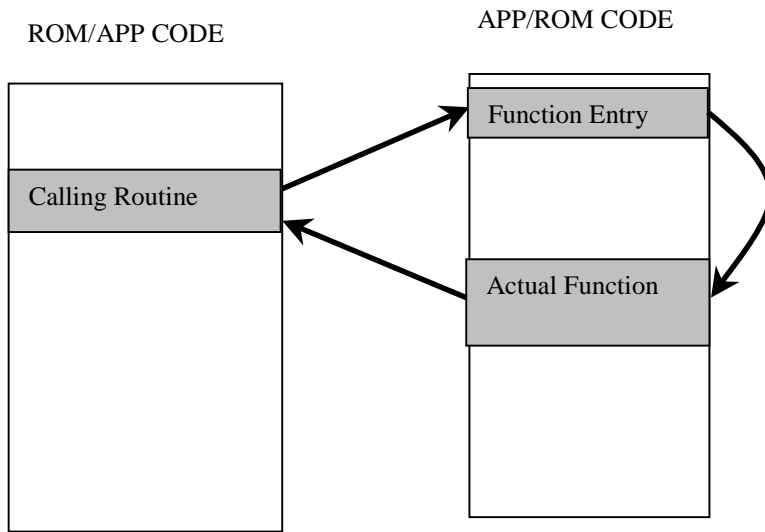
In the interrupt mode, the TAS 1020 application has to provide routines to support the following main tasks:

- Device Record an Related Hardware Programming Handler: Handle the device profiles and program the related hardware if applied.
- Provide the miscellaneous ISR routines if applied.

5 TAS 1020 Application/ROM Code Interface

Generally, the TAS 1020 application and ROM code provide a fixed location function. The parameters exchanged partly through the fixed external shared RAM and partly, the command code and/or the shared RAM data pointer, through the internal data RAM. So the process for calling a routine is normally setting up the parameters if required, then calling the fixed location with the command code and if required the shared data pointer passed to it. In the fixed location, there is an assembly “jump” command to the actual routine. Figure xxx shows the calling model.

Figure xx



Besides the exchanged parameters, in case of the SET requests, the data if sent by the host PC is stored in the fixed location external RAM, USB_EP0_XFERDATA[].

For the TAS 1020 application, the fixed location function is set at 0x002B. The following assembly code , part of the TAS 1020 application, defines a fixed location function with a jump to the actual function in the TAS 1020 application, FunctionName(...):

```
EXTERN CODE (_FunctionName)
CSEG  AT      002BH
LJMP  _FunctionName
END
```

For the TAS 1020 ROM code, the entry point function is set at 0x9FFD. The following code, part of the TAS 1020 application, defines a function which a jump to the fixed location function in the TAS 1020 ROM code:

```
#define PUBLIC_ROM_FUNCTION_ENTRY      0x9FFD

Word devRomFunction(byte Cmd)
{
#pragma ASM
```

```
LJMP    PUBLIC_ROM_FUNCTION_ENTRY
#prama ENDASM
    return 1;
}
```

The next sections describe the actual functions.

5.1 TAS 1020 Application Interface Function

This function is provided by the application for the ROM code USB Protocol Module and miscellaneous ISR routines. The prototype: void FunctionName(byte Cmd, xdata void *DataPtr) where:

- Cmd is a command code.
- DataPtr is a pointer to the external shared data.

The passing and returned parameters are based on the command code. The appendix XXX lists command codes along with parameters used.

Following is an example. Note that the variable names are used here do not have a significant meaning other than an example.

```

Void FunctionName(byte Cmd, void xdata *DataPtr)
{
    switch(Cmd)
    {
        // Case of valid/supported read requests
        case CMD_GET_REQUEST_DATA1:
            ...
            RET_DATA_PTR = &Data1;
            Break;

        // Case of valid/supported write requests with host PC data stored in
        // the USB_EP0_XFERDATA[].
        // Data1 is byte size
        case CMD_SET_REQUEST_DATA1:
            Data1 = USB_EP0_XFERDATA[0] ;
            RET_DATA_BYTE = EVENT_OK;
            Break;

        // Data2 is word size:
        case CMD_SET_REQUEST_DATA2:
            Data2 = (USB_EP0_XFERDATA[0] << 8) | USB_EP0_XFERDATA[1];
            RET_DATA_BYTE = EVENT_OK;
            Break;

        // Case requests other than read or write
        case CMD_OTHER:
            ...
            RET_DATA_BYTE = EVENT_OK;
            RET_DATA_PTR = SomeAddress;
            break;

        // Case invalid or unsupported command: no code
        default:
            RET_DATA_BYTE = EVENT_ERROR;
            RET_DATA_PTR = 0;
            break;
    }
}

```

5.2 TAS 1020 ROM Code Function

This function is provided by the ROM code for the application. The prototype: void FunctionName(byte Cmd) where Cmd is the command code.

The passing and returned parameters are based on the command code and if required have to be setup in the external shared RAM. The appendix YYY lists command codes along with parameters used.

Following are examples how the application call ROM code utility function to setup the USB engine and the ROM DFU state machine:

```
// Using ROM USB engine
//It is required to set the max size of control endpoint 0 before calling the ROM function
USBENGINE_EP0_SIZE = 8;
DevRomFunction(ROM_ENG_USB_INIT);
...
// Using ROM DFU Handler: the ROM DFU state machine need to be setup
// by calling the ROM utility function
DevRomFunction(ROM_INIT_DFU_STATE);
```

6 TAS 1020 ROM Memory Map

The ROM code uses the PDATA memory section in the external RAM for global and local variables. It also uses part of the external RAM for the USB engine mainly for the Control Endpoint 0 and for storing data sent by host PC. As the application sets the size of the Control Endpoint 0 and has knowledge about the maximum size of data could be received from the host in one specific USB request, it is the task of the application to allocate memory for use after the fixed memory allocated for the global and local variables of the ROM code. Figure xx shows the ROM code memory map.

Address	Size	Type	Names	Description
0xFA00	17	PDATA	UsbRequest	USB request and variables for USB engine and USB protocol operations
0xFA11	9	PDATA	EngParms	Parameters for USB engine operations
0xFA1A	14	PDATA	DfuStatemachine	Global variables for ROM DFU state machine and operations
0xFA28	8	PDATA	DfuDevice	ROM DFU device record for DFU mode only
0xFA30	32	PDATA	Externaldata	Data exchanged for ROM/Application
0xFA50	64	PDATA Group	ROM global/local variables	Global and local variables used strictly by ROM code
0xFA90	TBD	PDATA/ External RAM	INPACK_ADDR	Buffer for Control Endpoint 0 IN PIPE
TBD	TBD	PDATA/ External RAM	OUTPACK_ADDR	Buffer for Control Endpoint 0 OUT PIPE
TBD	TBD	PDATA/ External RAM	USB_EP0_XFERDATA	Buffer for storing data of the USB request if any.

TBD: To be defined by the Application.

As the ROM code uses only the PDATA for operations, the Application can use the whole 256 bytes of internal and indirect RAM. The code for initialize the PDATA section for ROM code is as following:

P2 = 0xFA;

This code has to be run before the Application can use the ROM code supports.

7 Application Implementation

This section is an example for writing the Application. It includes 3 parts:

- Initialize for ROM operations.
- Writing Device Record and Related Hardware Handler to support the ROM USB Protocol Handler and Hooks for miscellaneous ISRs and ROM Unsupported Feature.
- Required Routines and PDATA defines.

Note that the specific part related to the Application is not discussed here: device descriptors, initialize device record, programming Codecs, programming STC for application specific operations, ...

The code for demo is taken from the example application source code

7.1 Initialization

The initialization for ROM operation is as following:

```
Void main()
{
    // Take STC out of sleep mode: a safe practice
    GLOBCTL |= 0x04;

    // Set Port 2 for PDATA addressing by ROM code
    P2 = 0xFA;

    // Do device specific initialization
    ....

    // Initialize USB Engine: set Control Endpoint size to 8
    // and ROM utilities
    USBENGINE_EP0_SIZE = 0x8;
    DevRomFunction(ROM_ENG_USB_INIT);

    // Optional: initialize ROM DFU state machine if device has DFU interface
    DevRomFunction(ROM_INIT_DFU_STATE);

    // Application specific code.
    ...
}
```

7.2 Device Record Handler

This routine receives command code, the USB protocol parsing information from the ROM code if applied, , and updates or retrieves the device data accordingly. In some case the related hardware is programmed as

required by the USB protocol or specific application requirement. Following is taken from the sample application with some modification for easy reading. Note that the xdata *Data pointer is not used as the Application can access directly to the external RAM. This approach reduces the code size.

```
Void DevFunctionParser(byte Cmd, void xdata *Data)
{
    switch(Cmd)
    {
        case DEV_USERINTHANDLER:
            ...
            // These lines may not be required
            RET_DATA_BYTE = EVENT_OK;
            RET_DATA_PTR = NULL;
            Break;

        // Get Configuration
        case DEV_GETCONFIG:
            // Case GET request, we only need to set the return data pointer
            RET_DATA_PTR = (byte *)&Device_Config_Setting;
            Break;

        // Set Configuration
        case DEV_SETCONFIG:
            // Case SET request, we only need to set the return data byte
            RET_DATA_BYTE = EVENT_OK;
            Device_Config_Setting = PARAMS_SETCONFIG_CONFIGID;
            If (Device_Config_Setting == 0)
            {
                // Do some thing here for configuration 0
            }
            else (Device_Config_Setting == 1)
            {
                // Do some thing here for configuration 1
            }
            else
            {
                // Other configuration values are not supported
                RET_DATA_BYTE = EVENT_ERROR;
            }
            break;
    }
}
```

```

// Set Device Remote Wake
case DEV_SETREOTEWAKEUP:
    Device_Status |= DEV_STATUS_REMOTE_WAKEUP;
    RET_DATA_BYTE = EVENT_OK;
    Break;

....

// Unsupported features or invalid command codes
default:
    // Here we set both values because we don't know if this is a GET request
    // ,a SET request, or others
    RET_DATA_BYTE = EVENT_ERROR;
    RET_DATA_PTR = 0;
    Break;
}

}

```

7.3 Required Routines and PDATA Defines

7.3.1 Required Routines

These routines are required for the Application to interface with the ROM code. They are written in assembly. The following are samples from the sample application code:

- Routine to call the USB engine ext 0 interrupt service routine:

```

#define ROM_EXTINT0_ADDR    0x8003
Void UseRomExtInt0() interrupt 0
{
#pragma    ASM

    push    IE
    call    ROM_EXTINT0_ADDR
    pop     IE

#pragma    ENDASM
}

```

- Routine to allow the Application to call ROM utilities:

```

#define PUBLIC_ROM_FUNCTION_ENTRY 0x9FFD
Word devRomFunction(byte Function)
{
#pragma    ASM
    LJMP    PUBLIC_ROM_FUNCTION_ENTRY
#pragma    ENDASM
    // Not used
    return 1;
}

```

```
}

```

- Routine to allow the ROM code to call the Application Device Record Handler, DevFunctionParser():

```

    EXTERN CODE(_DevFunctionParser)
    CSEG  AT      002BH
    LJMP  _DevFunctionParser
    END

```

7.3.2 Required PDATA Defines

These are the required defined PDATA which are mostly used by the ROM code.

USB_REQUEST_STRUCT	pdata UsbRequest	_at_	0xFA10;
ENG_PARM_STRUCT	pdata EngParms	_at_	0xFA21;
DFU_STRUCT	pdata DfuStatemachine	_at_	0xFA28;
DFU_DEVICE_STRUCT	pdata DfuDevice	_at_	0xFA38;
XDATA_STRUCT	pdata Externaldata	_at_	0xFA40;

As the Application now know exactly where is the shared data Externaldata, in devFunctionParser(byte Cmd, void xdata *Data), the *Data pointer which points to the Externaldata is not used. This reduces the code size.

8 Appendix A

8.1 General commands

DEV_USERINTHANDLER

Value : 1

Params :

PARAMS_USERINTHANDLER_INTVECTOR, Source interrupt vector.

Return : None

Description: When receiving this command, the application will base on the source interrupt vector and perform the appropriate task.

DEV_CLASSHANDLER

Value : 58

Params :

PARAMS_CLASSHANDLER_USBREQ_PTR, pointer to the USB request.

Return : One of the following

Result in byte, EVENT_OK or EVENT_ERROR for set request.

Pointer to the data to be sent to host for get request.

Description: The application receives this command when the USB request is of type Class and the ROM code does not support this request.

8.2 ROM Support commands

DEV_GETIFCLASS

Value : 2

Params :

PARAMS_GETIFCLASS_IFID, interface id.

Output :

Return : Value defined the classes in word.

Description: Base on the interface id, the application will return the related class for that interface.

DEV_GETEPCLASS

Value : 3

Params :

PARAMS_GETEPCLASS_EPID, endpoint id.

Output :

Return : Value defined the classes in word.

Description: Base on the interface id, the application will return the related class for that endpoint.

DEV_USBVENDORHANDLER

Value : 4

Params :

- PARAMS_USBVENDORHANDLER_USBREQ_PTR, pointer to the USB request.

Output :

Return : Return one of the followings depending on the Vendor Request.

- Result in byte: EVENT_OK or EVENT_ERROR for USB SET requests.
- Pointer to the data to be sent to host for USB GET requests.

Description: When receiving this command, the application will decode the USB request and acts accordingly.

DEV_USBSTANDARDHANDLER

Value : 5

Params :
Output :
Return : None
Description: Not used.

8.3 USB Standard Chapter 9

8.3.1 Get Request Commands

DEV_GETCONFIG

Value : 6
Params : None
Return : Pointer to the configuration setting data.
Description: Get Configuration request.

DEV_GETDEVICESTATUS

Value : 7
Params : None
Return : Pointer to the device status data.
Description: Get device status.

DEV_GETIFSTATUS

Value : 8
Params :
PARAMS_GETIFSTATUS_IFID, interface ID.
Return : Pointer to the interface status data.
Description: Get interface status request.

DEV_GETEPSTATUS

Value : 9
Params :
PARAMS_GETEPSTATUS_EPID, endpoint ID.
Return : Pointer to the endpoint status data.
Description: Get endpoint status request.

DEV_GETDEVDESC

Value : 10
Params : None
Return : Two values.
PARAMS_GETDEVDESC_SIZE, The size of the device descriptor.
Pointer tot the device descriptor.
Description: Get device descriptor.

DEV_GETCONFIGDESC

Value : 11
Params :
PARAMS_GETCONFIGDESC_CONFIGID, Configuration ID.
Return : Two values.

- PARAMS_GETCONFIGDESC_SIZE, The size of the device descriptor.
- Pointer to the configuration descriptor block.

Description: Get Configuration descriptor.

DEV_GETHIDDESC

Value : 12

Params : PARAMS_GETHIDDESC_IFID, the HID interface ID.

Return : Two values.

- PARAMS_GETHIDDESC_SIZE, The size of the HID specific interface descriptor.
- Pointer to the HID specific interface descriptor.

Description: Get HID specific interface descriptor.

DEV_GETHIDREPORTDESC

Value : 13

Params :

PARAMS_GETHIDREPORTDESC_IFID, the HID interface ID.

Return : Two values.

- PARAMS_GETHIDREPORTDESC_SIZE, the size of the Hid Report descriptor.
- Pointer to the HID Report descriptor.

Description: Get HID Report descriptor.

DEV_GETSTRDESC

Value : 14

Params : PARAMS_GETSTRDESC_STRID, String ID.

Return : Two values.

- PARAMS_GETSTRDESC_SIZE, the size of the string descriptor.
- Pointer to the string descriptor.

Description: Get string descriptor.

DEV_GETIF

Value : 15

Params : PARAMS_GETIF_IFID, interface ID.

Return : Pointer to the current interface setting data.

Description: Get interface current setting request.

8.3.2 Set Request Commands

DEV_SETREMOTEWAKEUP

Value : 16

Params : PARAMS_SETEPFEATURE_EPID, endpoint ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR

Description: Set Remote Wake Up for the device.

DEV_SETEPFEATURE

Value : 17

Params : PARAMS_SETEPFEATURE_EPID, endpoint ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set endpoint feature request with endpoint stall feature.

DEV_SETCONFIG

Value : 18

Params : PARAMS_SETCONFIG_CONFIGID, configuration ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set configuration.

DEV_SETIF

Value : 19

Params :

- PARAMS_SETIF_IFID: interface ID.
- PARAMS_SETIF_SETTING, interface setting value.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set interface setting.

DEV_CLRREMOTEWAKEUP

Value : 20

Params : None.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Clear remote wakeup to the device.

DEV_CLEAREPFEATURE

Value : 21

Params : PARAMS_CLEAREPFEATURE_EPID, endpoint ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Clear endpoint stall.

8.4 USB Audio Class

8.4.1 Set Request Commands

DEV_SETMIXERALL

Value : 22

Params :

Return :

Description: Not implemented as DEV_SETMIXER supports all mixer forms of parameter blocks.

DEV_SETMIXER

Value : 23

Params :

- PARAMS_SETMIXER_INPUT, input mixer channel ID.
- PARAMS_SETMIXER_OUTPUT, output mixer channel ID.

Mixer gain values, 16 bit size, are in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set mixer gain request.

DEV_SETMUTE

Value : 24

Params :

- PARAMS_SETMUTE_UNITID, Feature Unit ID.
- PARAMS_SETMUTE_CN, channel number.
- Feature mute control value, boolean (byte), is in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Mute Control using the first form of parameter block.

DEV_SETMUTEALL

Value : 25

Params :

- PARAMS_SETMUTEALL_UNITID, Feature Unit ID.
- PARAMS_SETMUTEALL_NUMCN, number of channels to be programmed starting from channel 1.
- Feature mute control values, boolean (byte), are in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Mute Control using the second form of parameter block.

DEV_SETVOL

Value : 26

Params :

- PARAMS_SETVOL_UNITID, Feature Unit ID.
- PARAMS_SETVOL_CN, channel number.
- Feature volume control value, 16 bit size, is in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Volume Control using the first form of parameter block.

DEV_SETBASS

Value : 27

Params :

- PARAMS_SETBASS_UNITID, Feature Unit ID.
- PARAMS_SETBASS_CN, bass control id.
- Feature bass control value, 8 bit size, is in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Bass Control using the first form of parameter block.

DEV_SETTREBLE

Value : 28

Params :

- PARAMS_SETTREBLE_UNITID, Feature Unit ID.
- PARAMS_SETTREBLE_CN, treble control id.
- Feature treble control value, 8 bit size, is in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Treble Control using the first form of parameter block.

DEV_SETVOLALL

Value : 29

Params :

- PARAMS_SETVOLALL_UNITID, Feature Unit ID.
- PARAMS_SETVOLALL_NUMCN, number of channels to be programmed starting from channel 1.
- Feature volume control values, 16 bit size, are in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Volume Control using the second form of parameter block.

DEV_SETBASSALL

Value : 30

Params :

- PARAMS_SETBASSALL_UNITID, Feature Unit ID.
- PARAMS_SETBASSALL_NUMCN, number of controls to be programmed starting from control 1.
- Feature bass control values, 8 bit size, are in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Bass Control using the second form of parameter block.

DEV_SETTREBLEALL

Value : 31

Params :

- PARAMS_SETTREBLEALL_UNITID, Feature Unit ID.
- PARAMS_SETTREBLEALL_NUMCN, number of controls to be programmed starting from control 1.
- Feature treble control values, 8 bit size, are in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Feature Treble Control using the second form of parameter block.

DEV_SETFREQ

Value : 32

Params :

- PARAMS_SETFREQ_EPID, endpoint ID.
- Frequency values, three bytes, is in control endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set Endpoint request, sampling frequency.

8.4.2 Get Request Commands

DEV_GETUNITIDTYPE

Value : 33

Params : PARAMS_GETUNITIDTYPE_UNITID, Unit ID.

Return : The Audio Class-Specific Audio Control interface descriptor subtype value in byte.

Description: When receiving this command and the unit id, the application returns the related subtype of the addressed unit.

DEV_GETMIXER

Value : 34

Params :

- PARAMS_GETMIXER_INPUT, mixer input value.
- PARAMS_GETMIXER_OUTPUT, mixer output value.
- PARAMS_GETMIXER_TYPEVAL, request type value.
- PARAMS_GETMIXER_WLENGTH, length of the parameter block.

Return : Pointer to the buffer containing mixer values.

Description: Get mixer gain request.

DEV_GETMUTE

Value : 35

Params :

- PARAMS_GETMUTE_UNITID, Feature unit ID.
- PARAMS_GETMUTE_CN, the channel number.

Return : Pointer to the mute data.

Description: Get Feature Mute Control request.

DEV_GETVOL

Value : 36

Params :

- PARAMS_GETVOL_UNITID, Feature unit ID.
- PARAMS_GETVOL_CN, the channel number.
- PARAMS_GETVOL_TYPEVAL, request type value.

Return : Pointer to the channel volume data.

Description: Get Feature Volume Control request, first form of parameter block.

DEV_GETBASS

Value : 37

Params :

- PARAMS_GETBASS_UNITID, Feature unit ID.
- PARAMS_GETBASS_CN, the control number.
- PARAMS_GETBASS_TYPEVAL, request type value.

Return : Pointer to the data of the addressed bass control.

Description: Get Feature Bass Control request, first form of parameter block.

DEV_GETTREBLE

Value : 38

Params :

- PARAMS_GETTREBLE_UNITID, Feature unit ID.
- PARAMS_GETTREBLE_CN, the control number.
- PARAMS_GETTREBLE_TYPEVAL, request type value.

Return : Pointer to the data of the addressed treble control.

Description: Get Feature Treble Control request, first form of parameter block.

DEV_GETMIXERALL

Value : 39

Params :

return :

Description: Not implemented as DEV_GETMIXER handles all forms of parameter block.

DEV_GETMUTEALL

Value : 40

Params :

- PARAMS_GETMUTEALL_UNITID, Feature unit ID.
- PARAMS_GETMUTEALL_NUMCN, number of channels starting from channel 1.

Return : Pointer to a buffer containing mute values for the channels.

Description: Get Feature Mute Control with second form of parameter block.

DEV_GETVOLALL

Value : 41

Params :

- PARAMS_GETVOLALL_UNITID, Feature unit ID.
- PARAMS_GETVOLALL_NUMCN, number of channels starting from channel 1.
- PARAMS_GETVOLALL_TYPEVAL, request type value.

Return : Pointer to a buffer containing volume values for the channels.

Description: Get Feature Volume Control with second form of parameter block.

DEV_GETBASSALL

Value : 42

Params :

Return : None

Description: N/A

DEV_GETTREBLEALL

Value : 43

Params :

Return : None

Description: N/A

DEV_GETFREQ

Value : 44

Params : PARAMS_GETFREQ_EPID, endpoint ID.

Return : Pointer to the current endpoint frequency data, three bytes.

Description: Get Endpoint request, sampling frequency.

8.5 USB HID Class

DEV_HIDGETREPORT

Value : 45

Params :

- PARAMS_HIDGETREPORT_IFID, interface ID.
- PARAMS_HIDGETREPORT_REPORTTYPE, HID report type.
- PARAMS_HIDGETREPORT_REPORTID, HID report ID.

Return : Pointer to the related HID report value.

Description: Get HID report request.

DEV_HIDGETIDLE

Value : 46

Params : PARAMS_HIDGETIDLE_IFID, interface ID.

Return : Pointer to the HID Idle value.

Description: Get HID Idle request.

DEV_HIDGETPROTO

Value : 47

Params : PARAMS_HIDGETPROTOCOL_IFID, interface ID.

Return : Pointer to the HID protocol value.

Description: Get HID Protocol request.

DEV_HIDSETREPORT

Value : 48

Params :

- PARAMS_HIDSETREPORT_IFID, interface ID.
- PARAMS_HIDSETREPORT_REPORTTYPE, HID report type.
- PARAMS_HIDSETREPORT_REPORTID, HID report ID.
- Report value is stored in the Control Endpoint receive buffer.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set HID report request.

DEV_HIDSETIDLE

Value : 49

Params : PARAMS_HIDSETIDLE_IFID, interface ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set HID Idle request.

DEV_HIDSETPROTO

Value : 50

Params : PARAMS_HIDSETPROTOCOL_IFID, interface ID.

Return : Result in byte, EVENT_OK or EVENT_ERROR.

Description: Set HID Protocol request.

8.6 USB DFU Class

DFU get status command when App in IDLE or DETACH mode

DEV_DFUGETSTATUS

Value : 55

Params :

PARAMS_DFUGETSTATUS_STATE, DFU state.

PARAMS_DFUGETSTATUS_STATUS, DFU status.

Return : Pointer to DFU status data, DFU specs.

Description: DFU get status command when App in IDLE or DETACH mode. When receiving this command, base on the DFU state and status, the application returns pointer to the status data which created by the application.

.

DEV_DFUDNLOAD

Value : 51

Params :

- PARAMS_DFUDNLOAD_LEN, length on the data to be downloaded.
- PARAMS_DFUDNLOAD_PDATA_PTR, pointer to the data buffer.

Return : Three values:

- DEV_DFU_USB_STATUS: status of the operation, EVENT_OK or EVENT_ERROR.
- DEV_DFU_STATUS. DFU status, DFU Specs.
- DEV_DFU_LOAD_STATUS: state of download.

Description: This command is used for DFU mode with target is DFU_TARGET_OTHER. When receiving this command, the application will use the data as it likes and returns the requested values as appropriately.

DEV_DFUUPLOAD

Value : 52

Params :

- PARAMS_DFUUPLOAD_LEN, requested data length.
- PARAMS_DFUUPLOAD_PDATA_PTR, pointer to the data buffer.

Return :

- DEV_DFU_USB_STATUS: status of the operation, EVENT_OK or EVENT_ERROR.
- DEV_DFU_STATUS. DFU status, DFU Specs.
- DEV_DFU_XFER_LENGTH, the actual length of data to be uploaded.

Description: This command is used for DFU mode with target is DFU_TARGET_OTHER. When receiving this command, the application will store the related data as it likes to the buffer provided by the ROM code, and returns the requested values as appropriately.

DEV_DFUDNLOAD_START

Value : 53

Params : None

Return : None

Description: This command is used for DFU mode with target is DFU_TARGET_OTHER. This command is used to inform the application that the down load process is started. When receiving this command, the application will setup thing ready for the down load process.

DEV_DFUUPLOAD_START

Value : 54

Params : None

Return : None

Description: This command is used for DFU mode with target is DFU_TARGET_OTHER. This command is used to inform the application that the up load process is started. When receiving this command, the application will setup thing ready for the up load process.

DEV_DFUCHKMANIFEST

Value : 56

Params : None

Return : DEV_DFU_MNFSTATE, the state of the DFU manifestation.

Description: This command is used for DFU mode with target is DFU_TARGET_OTHER. This command is used by the ROM code to ask the application for the manifestation state. When receiving this command, the application returns with the state of the manifestation.

DEV_USERSUSPENSE

Value : 57

Params :

Return : None

Description: This command is used when the device is in DFU mode and the target is not RAM. The ROM sends this command to inform the application that the device has received a suspense request from host.

When receiving this command, the application should turn all the device specific hardware to sleep mode including the MCU.

9 Appendix B