



Reference Guide

SDK Documentation

Version 3.0
December 13, 2022

Table of contents

1. About This Guide	3
2. Introduction	3
3. HID Request Supported	3
4. HID Descriptors Supported	3
5. HID Connection	4
7. HID Communication	8
8. Commands Format	10
About the Message ID	12
Accessible Registers and commands	12
<i>I2C Commands</i>	12
<i>SPI Commands</i>	13
<i>FPGA Commands</i>	15
<i>FX3 Commands</i>	16
<i>UART Commands</i>	16
<i>Descriptor Commands</i>	17
<i>FX3 Registers</i>	18
9. Devices Version	19
10. Firmware Update	22
11. Bootloader	43
12. Troubleshooting	44

1. About This Guide

This document will help software developers to integrate the INOGENI HID interface to their software. Using the HID interface software developer will be able to update the INOGENI FX3, FPGA and EDID in the field.

An application source code in VB.NET/C/C++ and binaries are provided as examples.

2. Introduction

The INOGENI device is a composite USB device made of a UVC device, a UAC device and a HID device. The UVC function is used to send video from the device to the host. The UAC device is used to send audio to the host. The HID interface is used to send / receive commands between the device and the host. Both functions do not need a custom driver to work. The UVC, UAC and HID functions are using the built-in driver of popular operating systems like Windows, macOS and Linux. This document describes all the details about the HID function only.

To update the device software, we need to update the binary code for the FPGA, FX3 and EDID inside the device EEPROM. Once the binary in the EEPROM have been updated, a reset of the device is necessary so the device use the newly updated binary.

3. HID Request Supported

This HID device is supporting only one HID report: Report_ID “0”. **Any attempt to get or set to a Report_ID different than “0” will report an error.**

The following table list all HID class specific requests supported by this USB function. The HID function will answer a USB_STALL when an unsupported HID command is sent to the device (as specified inside the HID specification).

Table 1: Supported HID Requests

HID REQUESTS	VALUE ID	SUPPORTED
GET_REPORT	0x01	YES
GET_IDLE	0x02	NO
GET_PROTOCOL	0x03	NO
SET_REPORT	0x09	YES
SET_IDLE	0x0A	NO
SET_PROTOCOL	0x0B	NO

4. HID Descriptors Supported

The HID function reports only the following HID descriptors:

Table 2: HID Descriptors Supported

DESCRIPTORS	Comments
HID descriptor	HID version 1.01
Interface descriptor	-
Endpoint descriptor	Interrupt IN 64 bytes
Super Speed Endpoint Companion Descriptor	No burst
Report descriptor	64 bytes max

The HID function can handle only ONE command at a time. As long as the last issued command is not completed, the function will return a BUSY answer. For more details about the format of the commands, see the following section.

5. HID Connection

This section shows how to connect to the HID device.
INOGENI Vendor ID: 0x2997

6. Table 3: PID of INOGENI devices

DEVICE	PID
INOGENI DVIUSB	0x0001
INOGENI 4K2USB3	0x0004
INOGENI VGA2USB3	0x0009
INOGENI SDI2USB3	0x000B
INOGENI SHARE1	0x000D
INOGENI SHARE2	0x000E
INOGENI SHARE2 – REV2	0x000F
INOGENI HD2USB3	0x0010
INOGENI SHARE2U	0x0013
INOGENI SHARE1 – REV2	0x0014
INOGENI CAM300	0x0015
INOGENI CAM200	0x0016
INOGENI 4KXUSB3	0x0018
INOGENI SHARE2U – LX45	0x001A
INOGENI CAM300 – LX45	0x001B
INOGENI TOGGLE	0x001C
INOGENI UCAM	0x001E
INOGENI 4KX-PLUS	0x001F

Sample code

```

/*
API function: HidD_GetHidGuid
Get the GUID for all system HIDs.
Returns: the GUID in HidGuid.
*/

HidD_GetHidGuid(&HidGuid);

/*
API function: SetupDiGetClassDevs
Returns: a handle to a device information set for all installed devices.
Requires: the GUID returned by GetHidGuid.
*/

```

```

hDevInfo=SetupDiGetClassDevs
    (&HidGuid,
     NULL,
     NULL,
     DIGCF_PRESENT|DIGCF_INTERFACEDEVICE);

devInfoData.cbSize = sizeof(devInfoData);

//Step through the available devices looking for the one we want.
//Quit on detecting the desired device or checking all available devices without success.
do
{
    /*
    API function: SetupDiEnumDeviceInterfaces
    On return, MyDeviceInterfaceData contains the handle to a
    SP_DEVICE_INTERFACE_DATA structure for a detected device.
    Requires:
    The DeviceInfoSet returned in SetupDiGetClassDevs.
    The HidGuid returned in GetHidGuid.
    An index to specify a device.
    */

    Result=SetupDiEnumDeviceInterfaces
        (hDevInfo,
         0,
         &HidGuid,
         MemberIndex,
         &devInfoData);

    if (Result != 0)
    {
        //A device has been detected, so get more information about it.

        /*
        API function: SetupDiGetDeviceInterfaceDetail
        Returns: an SP_DEVICE_INTERFACE_DETAIL_DATA structure
        containing information about a device.
        To retrieve the information, call this function twice.
        The first time returns the size of the structure in Length.
        The second time returns a pointer to the data in DeviceInfoSet.
        Requires:
        A DeviceInfoSet returned by SetupDiGetClassDevs
        The SP_DEVICE_INTERFACE_DATA structure returned by SetupDiEnumDeviceInterfaces.

        The final parameter is an optional pointer to an SP_DEV_INFO_DATA structure.
        This application doesn't retrieve or use the structure.
        If retrieving the structure, set
        MyDeviceInfoData.cbSize = length of MyDeviceInfoData.
        and pass the structure's address.
        */

        //Get the Length value.
        //The call will return with a "buffer too small" error which can be ignored.

        Result = SetupDiGetDeviceInterfaceDetail
            (hDevInfo,
             &devInfoData,
             NULL,
             0,
             &Length,
             NULL);

        //Allocate memory for the hDevInfo structure, using the returned Length.
        detailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(Length);

        //Set cbSize in the detailData structure.
        detailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

        //Call the function again, this time passing it the returned buffer size.
        Result = SetupDiGetDeviceInterfaceDetail
            (hDevInfo,
             &devInfoData,
             detailData,
             Length,
             &Required,
             NULL);

        // Open a handle to the device.
        // To enable retrieving information about a system mouse or keyboard,
        // don't request Read or Write access for this handle.

        /*
        API function: CreateFile
        Returns: a handle that enables reading and writing to the device.
        Requires:
        The DevicePath in the detailData structure
        returned by SetupDiGetDeviceInterfaceDetail.
        */

        DeviceHandle=CreateFile

```

```

        (detailData->DevicePath,
        0,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        (LPSECURITY_ATTRIBUTES)NULL,
        OPEN_EXISTING,
        0,
        NULL);

    DisplayLastError("CreateFile: ");

    /*
    API function: HidD_GetAttributes
    Requests information from the device.
    Requires: the handle returned by CreateFile.
    Returns: a HIDD_ATTRIBUTES structure containing
    the Vendor ID, Product ID, and Product Version Number.
    Use this information to decide if the detected device is
    the one we're looking for.
    */

    //Set the Size to the number of bytes in the structure.

    Attributes.Size = sizeof(Attributes);

    Result = HidD_GetAttributes
        (DeviceHandle,
        &Attributes);

    DisplayLastError("HidD_GetAttributes: ");

    //Is it the desired device?

    MyDeviceDetected = FALSE;

    if ((Attributes.VendorID == VendorID) && (Attributes.ProductID == ProductID) || ((Attributes.VendorID ==
VendorID2) && (Attributes.ProductID == ProductID2)))
    {
        //Both the Vendor ID and Product ID match.

        MyDeviceDetected = TRUE;
        MyDevicePathName = detailData->DevicePath;
        DisplayData("Device detected");

        //Register to receive device notifications.

        RegisterForDeviceNotifications();

        //Get the device's capabilities.

        GetDeviceCapabilities();

        // Get a handle for writing Output reports.

        WriteHandle=CreateFile
            (detailData->DevicePath,
            GENERIC_WRITE,
            FILE_SHARE_READ|FILE_SHARE_WRITE,
            (LPSECURITY_ATTRIBUTES)NULL,
            OPEN_EXISTING,
            0,
            NULL);

        DisplayLastError("CreateFile: ");

        // Prepare to read reports using Overlapped I/O.

        PrepareForOverlappedTransfer();

    } //if (Attributes.ProductID == ProductID)
    else
        //The Product ID doesn't match.

        CloseHandle(DeviceHandle);

    } //if (Attributes.VendorID == VendorID)
    else
        //The Vendor ID doesn't match.

        CloseHandle(DeviceHandle);

    //Free the memory used by the detailData structure (no longer needed).

    free(detailData);

} //if (Result != 0)

else
    //SetupDiEnumDeviceInterfaces returned 0, so there are no more devices to check.

    LastDevice=TRUE;

```

```

        //If we haven't found the device yet, and haven't tried every available device,
        //try the next one.

        MemberIndex = MemberIndex + 1;

    } //do

    while ((LastDevice == FALSE) && (MyDeviceDetected == FALSE));

```

After the code above find the HID device we will be able to write 64 bytes report with the device handle.

Here is also a sample code that can be used with **libusb-1.0** library. This library is compatible with Windows, macOS and Linux.

```

#include "libusb-1.0/libusb.h"

int Connect(void)
{
    libusb_device **list = NULL;
    static const int INOGENI_VENDOR_ID = 0x2997;
    static const int INOGENI_VIDYO_PRODUCT_ID = 0x0002;
    static const int INOGENI_PRODUCT_ID = 0x0001;
    int device_ready = 0;
    int result;
    int rc = 0;
    size_t idx = 0;
    ssize_t count = 0;
    struct inogeni_hid_command command;
    int i = 0;

    rc = libusb_init(&context);
    if(rc != 0)
    {
        printf("error\n");
        return rc;
    }

    count = libusb_get_device_list(context, &list);
    if(!(count > 0))
    {
        printf("error\n");
        return LIBUSB_ERROR_NO_DEVICE;
    }

    for (idx = 0; idx < count; ++idx)
    {
        libusb_device *device = list[idx];

        struct libusb_device_descriptor desc = {0};
        rc = libusb_get_device_descriptor(device, &desc);
        if(rc != 0)
        {
            printf("error\n");
            return rc;
        }

        // printf("Vendor:Device = %04x:%04x\n", desc.idVendor, desc.idProduct);
        if ((desc.idVendor == INOGENI_VENDOR_ID) && ((desc.idProduct == INOGENI_VIDYO_PRODUCT_ID)
||
                (desc.idProduct == INOGENI_PRODUCT_ID)) )
        {
            // printf("INOGENI HDMI/DVI-2-USB3 device found!\n");
            result = libusb_open(device, &devh);
            if (result)
            {
                printf("Error opening the device = %X\n", result);
                return result;
            }
        }
    }
}

```

```

    }
  }
}

// Change these as needed to match idVendor and idProduct in your device's device descriptor.
if (devh != NULL)
{
  // The HID has been detected.
  // Detach the hidusb driver from the HID to enable using libusb.
  libusb_detach_kernel_driver(devh, INTERFACE_NUMBER);
  {
    result = libusb_claim_interface(devh, INTERFACE_NUMBER);
    if (result >= 0)
    {
      device_ready = 1;
    }
    else
    {
      fprintf(stderr, "libusb_claim_interface error %d\n", result);
    }
  }
}
else
{
  fprintf(stderr, "No INOGENI HDMI/DVI-2-USB3 device found!\n");
  return LIBUSB_ERROR_NO_DEVICE;
}
if (device_ready)
{
  return 0;
}

return LIBUSB_ERROR_NO_DEVICE;
}

```

7. HID Communication

This section shows how to communicate to the HID device.

To send a command to the USB3 dongle use the WriteFile:

```
Result = WriteFile (WriteHandle, OutputReport,
Capabilities.OutputReportByteLength, &BytesWritten, NULL);
```

The OutputReportByteLength is 64 bytes.

The content of OutputReport is the command sent to the Dongle, please see next section for more detail.

To get the answer to the command:

```
Result = HidD_GetInputReport(   ReadHandle,
                               InputReport,
                               Capabilities.InputReportByteLength);
```

This command is from the hid.dll.

The InputReportByteLength is 64 bytes like the output report length.

If you want to use the libusb-1.0 library, here are the basic functions for HID communication.

```
int SendReport(unsigned char *Data)
{
    int bytes_received;
    int bytes_sent;
    int i = 0;
    int result = 0;
    //char data_out[INOGENI_REPORT_LEN];

    // Send data to the device.
    bytes_sent = libusb_control_transfer(
        devh,
        CONTROL_REQUEST_TYPE_OUT ,
        HID_SET_REPORT,
        (HID_REPORT_TYPE_OUTPUT<<8)|0x00,
        INTERFACE_NUMBER,
        Data,
        INOGENI_REPORT_LEN,
        TIMEOUT_MS);

    if (bytes_sent >= 0)
    {
#ifdef DEBUG_DISPLAY_RX_TX
        printf("Output report data sent:\n");
        for(i = 0; i < 64; i++)
        {
            printf("%02x ",Data[i]);
        }
        printf("\n");
#endif //DEBUG_DISPLAY_RX_TX
    }
    else
    {
        fprintf(stderr, "Error sending Input report %d\n", result);
        return result;
    }

    return 0;
}
```

The Input Report Byte Length is also 64 bytes and he can be obtained from the dongle with for example the following code:

```
int ReceiveReport(unsigned char *Data)
{
    int bytes_received;
    int bytes_sent;
    int i = 0;
    int result = 0;

    // Request data from the device.
    bytes_received = libusb_control_transfer(
        devh,
        CONTROL_REQUEST_TYPE_IN ,
        HID_GET_REPORT,
        (HID_REPORT_TYPE_INPUT<<8)|0x00,
        INTERFACE_NUMBER,
        Data,
        INOGENI_REPORT_LEN,
        TIMEOUT_MS);

    if (bytes_received >= 0)
```

```

{
#ifdef DEBUG_DISPLAY_RX_TX
    printf("Input report data received:\n");
    for(i = 0; i < bytes_received; i++)
    {
        printf("%02x ", Data[i]);
    }
    printf("\n");
#endif //DEBUG_DISPLAY_RX_TX
}
else
{
    fprintf(stderr, "Error receiving Input report %d\n", result);
    return result;
}

return 0;
}

```

8. Commands Format

The Host exchange 64-bit report with the dongle. This section describes the message in exchange between the host and the dongle.

The following table indicates the general format of an HID command from the host to the dongle.

Table 4: General Command Format

BYTES INDEX	BYTES DESCRIPTION
0	CMD_INDEX
1	CMD_ID
2	DATA_LENGTH
3	DATA0 (OPTIONAL)
4	DATA1 (OPTIONAL)
5	...
...	...
63	DATA63 (OPTIONAL)

The answer receives from the dongle in case of success will have the following format:

Table 5: General Answer Format (Successful)

BYTES INDEX	BYTES DESCRIPTION
0	CMD_INDEX
1	CMD_ID
2	DATA_LENGTH

3	ACK = 0x00
4	DATA0 (OPTIONAL)
5	...
...	...
63	DATA62 (OPTIONAL)

However, if the command to the dongle failed you will receive the following answer type.

Table 6: General Answer Format (Fail)

BYTES INDEX	BYTES DESCRIPTION
0	CMD_INDEX
1	CMD_ID
2	DATA_LENGTH
3	ERROR_CODE

The following table presents all possible command ID.

Table 7: Commands ID

COMMANDS	ID (8 bits)
I2C_WR	0x00
I2C_RD	0x01
SPI_WR	0x02
SPI_RD	0x03
SPI_ERASE	0x04
SET_BOOTLOADER	0x05
FPGA_WR	0x06
FPGA_RD	0x07
DEBUG_WR	0x08
DEBUG_RD	0x09
UART_WR	0x0A
UART_RD	0x0B
SDI_WR	0x0C
SDI_RD	0x0D
DESCR_WR	0x0E
DESCR_RD	0x0F

Then the possible return code from the dongle.

Table 8: Error Code

BYTES INDEX	BYTES DESCRIPTION
0x00	ACK

0x01	BUSY
0x02	BAD_CMD_NUM
0x03	BAD_LENGTH
0x04	BAD_ADDRESS
0x05	TOO_BAD
0x06	BAD_DATA_LEN
0x07	SPI_FAIL
0x08	I2C_NO_ANSWER
0x09	I2C_COMM_FAILED
0X0A	NOT_SUPPORTED
0x0B	ACCESS_NOT_SUPPORTED
0x0C	FPGA_NOT_CONFIGURED

When the host received a busy, it should wait a little bit and try again.

About the Message ID

The message ID is 8 bits counter incremented at each new command sent by the host. The same index will be returned with the corresponding answer on the USB bus. The accepted range index from the host is 1-0xFF. The 0x00 code index is reserved for any “interrupt packets” used to signal a specific update from the device to the host.

Accessible Registers and commands

This section introduces all available command and register.

I2C Commands

Table 9: I2C Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x00
2	LENGTH	0x03-3A
3	I2C_ADDR	0 - 0xFF
4	I2C_REG	0 - 0xFF
5	I2C_DATA0	0 - 0xFF
...
n	I2C_DATA _n	0 - 0xFF

Table 10: I2C Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x01
2	LENGTH	0x03
3	I2C_ADDR	0 - 0xFF
4	I2C_REG	0 - 0xFF

5 I2C_LEN 0x01 – 0x3B

Table 11: I2C Read Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x01
2	LENGTH	0x02-0x3B
3	ACK	0x00
4	I2C_DATA	0 - 0xFF

SPI Commands

Table 12: SPI Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x02
2	LENGTH	0x05-0x3C
3	SPI_DEV	0x00
4	SPI_ADDR[16:23]	0 - 0xFF
5	SPI_ADDR[8:15]	0 - 0xFF
6	SPI_ADDR[0:7]	0 - 0xFF
7	SPI_DATA0	0 - 0xFF
...
n	SPI_DATA _n	0 - 0xFF

Table 13: SPI Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x03
2	LENGTH	0x05
3	SPI_DEV	0x00
4	SPI_ADDR[16:23]	0 - 0xFF
5	SPI_ADDR[8:15]	0 - 0xFF
6	SPI_ADDR[0:7]	0 - 0xFF
7	SPI_LEN	0x01-0x3A

Table 14: SPI Erase Sector Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x04
2	LENGTH	0x02
3	SPI_DEV	0x00

4

SECTOR

0 - 0xFF

Table 15: SPI Read Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x03
2	LENGTH	0x02-0x3C
3	ACK	0x00
4	SPI_DATA	0 - 0xFF

FPGA Commands

Table 16: FPGA Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x06
2	LENGTH	0x03-0x3D
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

Table 17: FPGA Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0
2	LENGTH	0x07
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	DATA_LEN	0x01-0x3A

Table 18: FPGA Read (0x09) and FPGA write (0x08) Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x06-0x07
2	LENGTH	0x02-3D
3	ACK	0x00
4	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

FX3 Commands

Table 19: Debug Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x08
2	LENGTH	0x03
3	ADDR	0 - 0xFF
4	DATA	0 - 0xFF

Table 20: Debug Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x09
2	LENGTH	0x01
3	REG	0 - 0xFF

Table 21: Debug Read Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x01
2	LENGTH	0x02
3	ACK	0x00
4	DATA	0x00 - 0xFF

UART Commands

Command set only available for SHARE2/SHARE2U and CAM series devices.

Table 22: UART Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0A
2	LENGTH	0x03-0x3D
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

Table 23: UART Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0B
2	LENGTH	0x07
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	DATA_LEN	0x01-0x3A

Table 24: UART Read (0x0B) and UART Write (0x0A) Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0A-0x0B
2	LENGTH	0x02-3D
3	ACK	0x00
4	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

Descriptor Commands

Descriptor commands allow you to burn a device description inside the embedded flash in order to easily identify the connected hardware.

Table 25: Descriptor Write Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0E
2	LENGTH	0x03-0x3D
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

Table 26: Descriptor Read Command

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0F
2	LENGTH	0x07
3	CONTROL	0 - 0xFF
4	ADDRESS	0 - 0xFF
5	DATA_LEN	0x01-0x3A

Table 27: Descriptor Read (0x0F) and Descriptor Write (0x0E) Answer

BYTES INDEX	BYTES DESCRIPTION	VALUE
0	CMD_INDEX	1 - 0xFF
1	CMD_ID	0x0E-0x0F
2	LENGTH	0x02-3D
3	ACK	0x00
4	FPGA_DATA0	0 - 0xFF
...
n	FPGA_DATA _n	0 - 0xFF

FX3 Registers

Table 28: FX3 Registers

Register Index	Description	Access	Device support
0x00	INOGENI Firmware Version Major	Read only	ALL
0x01	INOGENI Firmware Version Minor	Read only	ALL
0x02	EEPROM 0x00, 0x08 = Spartan-6 8Mb flash 0x10 = Spartan-6 16Mb flash 0x20 = Spartan-7	Read only	ALL
0x03	FPGA version	Read only	ALL
0x04	Reserved	Read/Write	ALL
0x05	First Video Width MSB	Read only	ALL
0x06	First Video Width LSB	Read only	ALL
0x07	First Video Height MSB	Read only	ALL
0x08	First Video Height LSB	Read only	ALL
0x09	First Video Frame Rate*100 MSB	Read only	ALL
0x0A	First Video Frame Rate*100 LSB	Read only	ALL
0x0B	First Video General	Read only	ALL
0x0C	EDID Version	Read only	ALL
0x0D	USB speed	Read only	ALL

0x0E	Second Video Width MSB	Read only	SHARE1 SHARE2
0x0F	Second Video Width LSB	Read only	SHARE1 SHARE2
0x10	Second Video Height MSB	Read only	SHARE1 SHARE2
0x11	Second Video Height LSB	Read only	SHARE1 SHARE2
0x12	Second Video Frame Rate*100 MSB	Read only	SHARE1 SHARE2
0x13	Second Video Frame Rate*100 LSB	Read only	SHARE1 SHARE2
0x14	Second Video General	Read only	SHARE1 SHARE2

The bit 0 of the General register is a video input valid. Before decoding the register 0x05 to 0x08 the host should verify that bit 0 is 1 to indicate valid video input. The bit 1 of the General register indicates if the FPGA is configured, for normal operation this bit must high. At boot up this bit will be 0, as the device is configured the bit will turn 1. The device cannot stream video when the bit is 0. The FPGA version major is bit 4 to 7 and the minor version is bit 0 to 3. Register 0x04 allow to disabled process and module in the FX3 and to check if the FPGA is active. The Register 0x02 is now obsolete.

9. Devices Version

All hardware in the INOGENI device have a version that can be retrieved from HID command. Before updating a device with a new firmware, bit stream or EDID it is good practice to verify the current version to determine if the device needs to be updated. If the device already has the good version, there is no need to proceed with the update.

It is also a good validation to read all the devices version after an update to verify that the update was successful. However, remember the device take the new firmware, bit stream and EDID only the next reboot after the update.

FX3 Version

The FX3 version is located at register 0x00 and 0x01. Those register can be get from the HID commands get FX3 register (see Table 20: Debug Read Command). The get FX3 register does not support the get of more than one register in one command, so to HID command are necessary to get the FX3 version.

Get the FX3 INOGENI Firmware Version Major:

TX: 01 09 01 00 00 00 ... 00 (total 64 bytes)

Command ID = 0x01 (any value from 0x01 to 0xFF)
Read FX3 register = 0x09

Len = 0x01
Register = 0x00 = FX3 version major register

Then wait for an answer:

RX: 01 09 02 00 01 00 ... 00 (total 64 bytes)

Command ID = 0x01, so this is an answer to our request
Read FX3 register = 0x09, so this is an answer to a get FX3 register
Len = 0x02 (ACK + Register value)
Register = 0x00 = ACK
DATA = FX3 major version = 0x01

Get the FX3 INOGENI Firmware Version Minor:

TX: 02 09 01 01 00 00 ... 00 (total 64 bytes)

Command ID = 0x02 (any value from 0x01 to 0xFF)
Read FX3 register = 0x09
Len = 0x01
Register = 0x01 = FX3 version minor register

Then wait for an answer:

RX: 02 09 02 00 35 00 ... 00 (total 64 bytes)

Command ID = 0x02, so this is an answer to our request
Read FX3 register = 0x09, so this is an answer to a get FX3 register
Len = 0x02 (ACK + Register value)
Register = 0x00 = ACK
DATA = FX3 minor version = 0x35

FPGA Version

The FPGA take 5 to 7 seconds longer to boot up than the FX3. The bit 1 in the FX3 Register (0x04) indicates if the FPGA has finish to boot up. You should not request the FPGA version until bit 1 is high.

Get FX3 General 0x04 register

TX: 58 09 01 04 00 ... 00 (total 64 bytes)

Command ID = 0x58 (any value from 0x01 to 0xFF)
Get FX3 Register command = 0x09
Len = 0x01
Register = 0x04 = the General register

The dongle answer:

RX: 58 09 02 00 02...

Command ID = 0x58 (the same than our request indicating an answer to that request...)
Get FX3 Register command = 0x09 (the same than our request indicating an answer to that request...)

Len = 0x02
Result Code = 0x00 = Acknowledge
Register value = 0x02 = the General register bit 1 is high then the FPGA is loaded!

The FPGA version is located at register FX3 register 0x03. That register can be get from the HID command get FX3 register (see Table 20: Debug Read Command).

Get the FPGA version:

TX: 03 09 01 03 00 00 ... 00 (total 64 bytes)

Command ID = 0x03 (any value from 0x01 to 0xFF)
Read FX3 register = 0x09
Len = 0x01
Register = 0x03 = FPGA version register.

Then wait for an answer:

RX: 03 09 02 00 01 00 ... 00 (total 64 bytes)

Command ID = 0x03, so this is an answer to our request
Read FX3 register = 0x09, so this is an answer to a get FX3 register
Len = 0x02 (ACK + Register value)
Register = 0x00 = ACK
DATA = 0x22 = FPGA Version 2.2

The FPGA version is on one byte, the 4 most significant bit is consider as the major and the 4 less significant bits is consider as the minor.

EDID Version

The EDID version is located at FX3 register 0x0C. That register can be get from the HID command get FX3 register (see Table 20: Debug Read Command).

Example: Get EDID version:

TX: 04 09 01 0C 00 00 ... 00 (total 64 bytes)

Command ID = 0x04 (any value from 0x01 to 0xFF)
Read FX3 register = 0x09
Len = 0x01
Register = 0x0C = EDID version

Then wait for an answer:

RX: 04 09 02 00 04 00 ... 00 (total 64 bytes)

Command ID = 0x04 (any value from 0x01 to 0xFF)
Read FX3 register = 0x09
Len = 0x02 (ACK + Register value)
Register = 0x00 = ACK
DATA = EDID version = 0x04 (EDID version 4)

10. Firmware Update

To update the INOGENI firmware, bit stream and EDID you need to update the binaries code in the device EEPROM. After updating the EEPROM with the new code, you will need to reboot the dongle before the firmware update take effect.

The previous sections show you how to access to the dongle, using those HID commands we will update the device firmware.

Before starting the software update of the device make sure it is not streaming video or audio. Before you can write a new binary in the device EEPROM, you need to erase the current content. See SPI Commands for more details.

The erase command is erasing a full sector at one time. A sector for the EEPROM is 64KB. The figures below indicate the sector use by the FX3, FPGA and the EDID depending on the hardware version you are running.

When probing the FX3 register 0x02, you can find the right mapping for binaries to program to the device.

Sector	Start Address	End Address	Content
0	000000	00FFFF	FX3
1	010000	01FFFF	
2	020000	02FFFF	
3	030000	03FFFF	FPGA
4	040000	04FFFF	
5	050000	05FFFF	
6	060000	06FFFF	
7	070000	07FFFF	
8	080000	08FFFF	
9	090000	09FFFF	
10	0A0000	0AFFFF	
11	0B0000	0BFFFF	
12	0C0000	0CFFFF	
13	0D0000	0DFFFF	Reserved
14	0E0000	0EFFFF	
15	0F0000	0FFFFFFF	EDID

Figure 1 - EEPROM Memory Mapping for Spartan-6 8Mb flash

Sector	Start Address	End Address	Content
0	000000	00FFFF	FX3
1	010000	01FFFF	
2	020000	02FFFF	
3	030000	03FFFF	FPGA#1 (Uncompressed)
4	040000	04FFFF	
5	050000	05FFFF	
6	060000	06FFFF	
7	070000	07FFFF	
8	080000	08FFFF	
9	090000	09FFFF	
10	0A0000	0AFFFF	
11	0B0000	0BFFFF	
12	0C0000	0CFFFF	
13	0D0000	0DFFFF	
14	0E0000	0EFFFF	Reserved
15	0F0000	0FFFFFFF	EDID
16	100000	10FFFF	FPGA#2 (MJPEG)
17	110000	11FFFF	
18	120000	12FFFF	
19	130000	13FFFF	
20	140000	14FFFF	
21	150000	15FFFF	
22	160000	16FFFF	
23	170000	17FFFF	
24	180000	18FFFF	
25	190000	19FFFF	
26	1A0000	1AFFFF	Reserved
27	1B0000	1BFFFF	
28	1C0000	1CFFFF	
29	1D0000	1DFFFF	
30	1E0000	1EFFFF	
31	1F0000	1FFFFFFF	Reserved

Figure 2 - EEPROM Memory Mapping for Spartan-6 16Mb flash

Sector	Start Address	End Address	Content	Sector	Start Address	End Address	Content
0	000000	00FFFF	FX3	32	200000	20FFFF	FPGA
1	010000	01FFFF		33	210000	21FFFF	
2	020000	02FFFF		34	220000	22FFFF	
3	030000	03FFFF	FPGA	35	230000	23FFFF	
4	040000	04FFFF		36	240000	24FFFF	
5	050000	05FFFF		37	250000	25FFFF	
6	060000	06FFFF		38	260000	26FFFF	
7	070000	07FFFF		39	270000	27FFFF	
8	080000	08FFFF		40	280000	28FFFF	
9	090000	09FFFF		41	290000	29FFFF	
10	0A0000	0AFFFF		42	2A0000	2AFFFF	
11	0B0000	0BFFFF		43	2B0000	2BFFFF	
12	0C0000	0CFFFF		44	2C0000	2CFFFF	
13	0D0000	0DFFFF		45	2D0000	2DFFFF	
14	0E0000	0EFFFF		46	2E0000	2EFFFF	
15	0F0000	0FFFFF		47	2F0000	2FFFFF	
16	100000	10FFFF		48	300000	30FFFF	
17	110000	11FFFF		49	310000	31FFFF	
18	120000	12FFFF		50	320000	32FFFF	
19	130000	13FFFF		51	330000	33FFFF	
20	140000	14FFFF		52	340000	34FFFF	
21	150000	15FFFF		53	350000	35FFFF	
22	160000	16FFFF		54	360000	36FFFF	
23	170000	17FFFF		55	370000	37FFFF	
24	180000	18FFFF		56	380000	38FFFF	
25	190000	19FFFF		57	390000	39FFFF	
26	1A0000	1AFFFF		58	3A0000	3AFFFF	
27	1B0000	1BFFFF		59	3B0000	3BFFFF	
28	1C0000	1CFFFF		60	3C0000	3CFFFF	
29	1D0000	1DFFFF		61	3D0000	3DFFFF	
30	1E0000	1EFFFF		62	3E0000	3EFFFF	Reserved
31	1F0000	1FFFFF		63	3F0000	3FFFFF	EDID

Figure 3 - EEPROM Memory Mapping for Spartan-7 32Mb flash

FX3 Firmware Update

Before Starting the update of the dongle EEPROM make sure that the dongle is not streaming video or audio. The new firmware for the FX3 is provided by INOGENI as an .img binary file. The FX3 is located in the first 3 sectors of the EEPROM. Those 3 sectors must be erased before any data can be written in the EEPROM. You can erase one sector in the EEPROM using the command found in Table 14: SPI Erase Sector Command.

Example of one command to erase the sector 2:

```
TX: 01 04 02 00 02 00 ... 00 (total 64 bytes)
```

Command ID = 0x01 (any value from 0x01 to 0xFF)
Erase sector command = 0x04
Len = 0x02
SPI dev = 0x00 (always 0x00 for the firmware update)
Sector = 0x02 = Erase sector 2 (last sector of FX3)

Then wait for an answer.

```
RX: 01 04 01 00 ... 00
```

Command ID = 0x01 same than our request, that tell us this an answer to our request.
Erase sector command = 0x04, so this answer to an erase sector.
Len = 0x01 so will follow: 1 result code
Result Code = 0x00 this an ACK

You must wait 3 seconds after each erase command to let the SPI EEPROM time to execute the command.

After you successfully erase the sector 0 to 2, you can start programming the FX3 in the EEPROM from the data of the .img file. The write of data to EEPROM is done using the command of Table 12: SPI Write Command.

Example of the command to program address 0x20100:

```
TX: 02 02 3C 00 02 01 00 XX ... YY (total 64 bytes)
```

Command ID = 0x02 (any value from 0x01 to 0xFF)
Write to EEPROM command = 0x02
Len = 0x3C
SPI dev = 0x00 (always 0x00 for the firmware update)
Address [23:16] (MSB) = 0x02
Address [15:8] = 0x01
Address [7:0] (LSB) = 0x00
Address = 0x20100
XX ... YY = Data 56 bytes

Then wait for an answer.

```
RX: 02 02 01 00 ... 00
```

Command ID = 0x02 same than our request, that tell us this an answer to our request.
Write to EEPROM command = 0x02, so this an answer to a write SPI
Len = 0x01 so will follow: 1 result code
Result Code = 0x00 this an ACK

So you need to program the whole .img file data into the EEPROM starting at address 0x00000 until the end of the file. You should always use to send write command of the maximum size of 56 bytes, except for the last write command. The figure below illustrates the flow chart for the FX3 update.

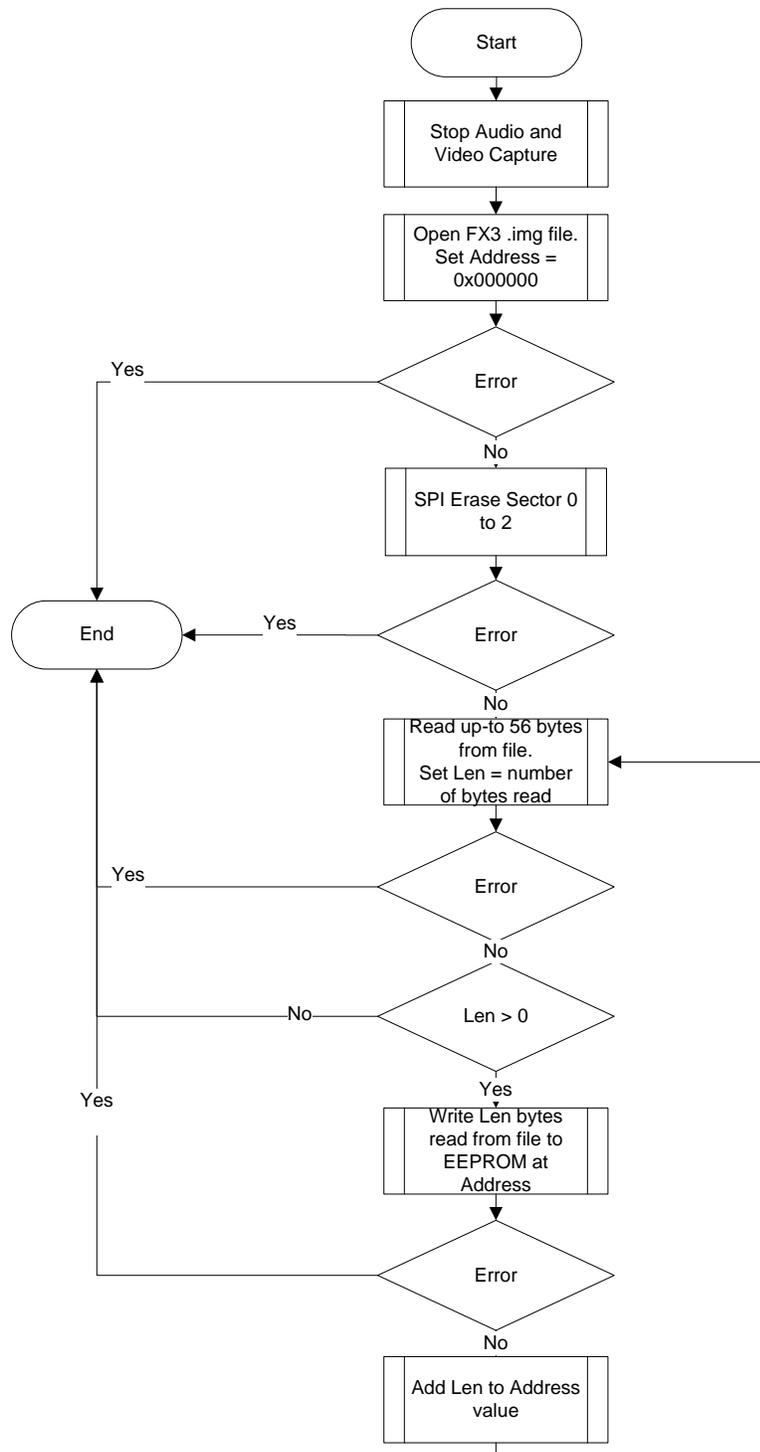


Figure 4 - FX3 Update Flow

To complete the FX3 firmware update a reboot will be necessary. However before performing a reboot you should complete the update of all part of the dongle first. If the FX3 reboots with a corrupted EEPROM it will boot in bootloader mode (see Bootloader section).

FPGA Bit stream Update

Before Starting the update of the dongle EEPROM make sure that the dongle is not streaming video or audio. We also need to stop the FX3 FPGA auto-configuration feature before we start updating the EEPROM with the new bit stream. This can be done by setting one bit in the FX3 General register 0x04. However, as we want to affect only one bit we start by getting the value of the register (see Table 20: Debug Read Command):

Get FX3 General 0x04 register

```
TX: 59 09 01 04 00 ... 00 (total 64 bytes)
```

Command ID = 0x59 (any value from 0x01 to 0xFF)

Get FX3 Register command = 0x09

Len = 0x01

Register = 0x04 = the General register

The dongle answer:

```
RX: 59 09 02 00 02 ...
```

Command ID = 0x59 (the same than our request indicating an answer to that request...)

Get FX3 Register command = 0x09 (the same than our request indicating an answer to that request...)

Len = 0x02

Result Code = 0x00 = Acknowledge

Register value = 0x02 = the General register

To stop the FPGA download we need to set bit 0 and leave all other bit at the same value:

Register General must be = 0x03

Let's do a set FX3 register command to set the register to this new value (see Table 19: Debug Write Command):

Set FX3 General Register (0x04) to 0x03

```
TX: 5A 08 02 04 03 ... 00 (total 64 bytes)
```

Command ID = 0x5A (any value from 0x01 to 0xFF)

Set FX3 Register command = 0x08

Len = 0x02

Register = 0x04 = the enable register

Register value = 0x03

Then wait for an answer.

```
RX: 5A 08 01 00 ... 00
```

Command ID = 0x5A same than our request, that tell us this an answer to our request.

Set FX3 Register command = 0x08, so this answer to an Set FX3 register.

Len = 0x01 so will follow: 1 result code

Result Code = 0x00 this an ACK

The FPGA bit stream for the device is provided by INOGENI binary files. The FPGA is located in:

- sector from 3 to 13 of the EEPROM (See Figure 1: EEPROM Memory Mapping for Spartan-6 8Mb flash) for Spartan-6 8Mb version.
FPGA_START_ADDRESS = 0x30000
- sector from 3 to 13 (FPGA#1) and sector 16 to 26 (FPGA #2) (See Figure 2: EEPROM Memory Mapping for Spartan-6 16Mb flash) for Spartan-6 16Mb version.
 - o FPGA #1: uncompressed FPGA binary for use over USB3.
 - FPGA_START_ADDRESS = 0x30000
 - o FPGA #2: MJPEG FPGA binary for use over USB2.
 - FPGA_START_ADDRESS = 0x100000
- sector from 3 to 60 of the EEPROM (See Figure 3: EEPROM Memory Mapping for Spartan-7 32Mb flash) for Spartan-7 32Mb version.
 - o FPGA_START_ADDRESS = 0x30000

The right mapping to use is defined in the FX3 register 0x02.

Every sector that will be used to hold the new bit stream must be erased before any data can be written in the EEPROM. You can erase one sector using the command found in Table 14: SPI Erase Sector Command.

Example of one command to erase the sector 3:

```
TX: 01 04 02 00 03 00 ... 00 (total 64 bytes)
```

```
Command ID = 0x01 (any value from 0x01 to 0xFF)
Erase sector command = 0x04
Len = 0x02
SPI dev = 0x00 (always 0x00 for the firmware update)
Sector = 0x03 = Erase sector 3 (first sector for the FPGA)
```

Then wait for an answer.

```
RX: 01 04 01 00 ... 00
```

```
Command ID = 0x01 same than our request, that tell us this an answer to our request.
Erase sector command = 0x04, so this answer to an erase sector.
Len = 0x01 so will follow: 1 result code
Result Code =0x00 this an ACK
```

You must wait 3 seconds after each erase command to let the SPI EEPROM time to execute the command.

After you successfully erase the FPGA sectors, you need to write FPGA bit stream size with a checksum at address FPGA_START_ADDRESS. The size of the FPGA bit stream is equal to the .bin file size, so the number of byte inside the bit file. The size of the bit stream is an unsigned on 32-bit. The checksum is on one byte, so the five first byte of the FPGA section are reserved for the size. Additionally, the first 256 byte (0x00-0xFF) are reserved for the bit stream size, so the next 251 byte are left empty:

Address	Content
0x30000	Size [0:7] (LSB)
0x30001	Size [8:15]
0x30002	Size [16:23]
0x30003	Size [24:31] (MSB)
0x30004	Check Sum
0x30005	Blank
...	
...	
0x30100	Start Of the bit stream
...	...
...	...

Figure 5 - FPGA EEPROM Size Section

The size must be written to the 4 first byte of the sector:

```
buffer[0] = (u8)(FileSize & 0xFF);
buffer[1] = (u8)((FileSize & 0xFF00) >> 8);
buffer[2] = (u8)((FileSize & 0xFF0000) >> 16);
buffer[3] = (u8)((FileSize & 0xFF000000) >> 24);
```

The checksum is computed from the 4 byte of the size:

```
for (i = 0; i < 4; i++)
{
    checksum += buffer[i];
}
checksum = ((256 - checksum) % 256);
buffer[4] = checksum;
```

Then the size can be written in the EEPROM using the write SPI command at address `FPGA_START_ADDRESS`:

`TX: 0A 02 3C 00 03 00 00 AA BB CC DD EE 00 ... 00 (total 64 bytes)`

Command ID = 0x0A (any value from 0x01 to 0xFF)
Write to EEPROM command = 0x02
Len = 0x3C
SPI dev = 0x00 (always 0x00 for the firmware update)
Address [23:16] (MSB) = 0x03
Address [15:8] = 0x00
Address [7:0] (LSB) = 0x00
Address = 0x30000
AA = Buffer[0]
BB = Buffer[1]
CC = Buffer[2]
DD = Buffer[3]
EE = Buffer[4]

Then wait for an answer.

RX: 0A 02 01 00 ... 00

Command ID = 0x0A same than our request, that tell us this an answer to our request.
Write to EEPROM command = 0x02, so this an answer to a write SPI
Len = 0x01 so will follow: 1 result code
Result Code =0x00 this an ACK

You can start programming the FPGA in the EEPROM from the data of the bit file. You start with the first 56 bytes of the file at address `FPGA_START_ADDRESS + 0x100`. The write of data to EEPROM is done using the command of Table 12: SPI Write Command.

Example: Write the first 56 bytes at address `FPGA_START_ADDRESS + 0x100`:

TX: 02 02 3C 00 03 01 00 XX ... YY (total 64 bytes)

Command ID = 0x02 (any value from 0x01 to 0xFF)
Write to EEPROM command = 0x02
Len = 0x3C
SPI dev = 0x00 (always 0x00 for the firmware update)
Address [23:16] (MSB) = 0x03
Address [15:8] = 0x01
Address [7:0] (LSB) = 0x00
Address = 0x30100
XX ... YY = Data 56 bytes

Then wait for an answer.

RX: 02 02 01 00 ... 00

Command ID = 0x02 same than our request, that tell us this an answer to our request.
Write to EEPROM command = 0x02, so this an answer to a write SPI
Len = 0x01 so will follow: 1 result code
Result Code =0x00 this an ACK

So you need to program the whole bit file data into the EEPROM starting at address `FPGA_START_ADDRESS + 0x100` until the end of the file. You should always use to send write command of the maximum size of 56 bytes, except for the last write command. The figure below illustrates the flow chart for the FPGA update.

You also need to do this procedure for the second FPGA binary (FPGA#2) if this is applicable to your device.

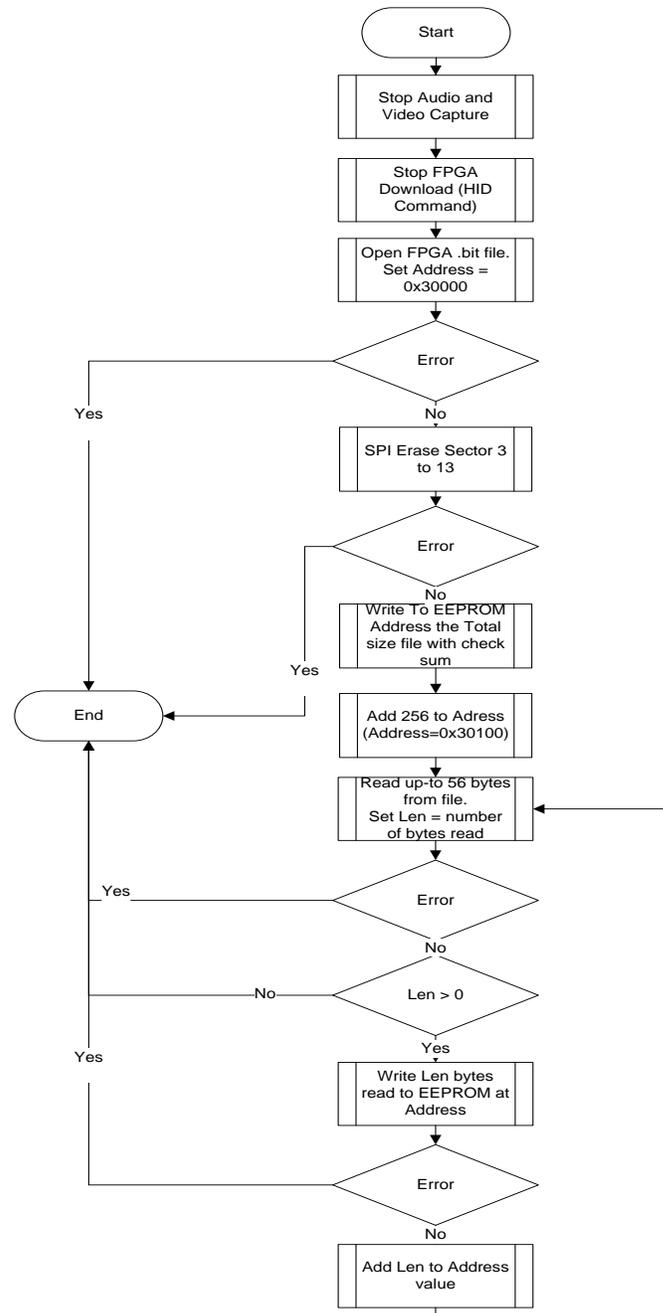


Figure 6 - FPGA Update Flow

To complete the FPGA bit stream update a reboot will be necessary. However before performing a reboot you should complete the update of all part of the dongle first. Note that a get FX3 version command before rebooting will not return the new version number as the FX3 is still using the previous code, as it is loaded into RAM memory at boot up by the device bootloader.

EDID Update

The FX3 dynamically modify bytes in the EDID, as result you cannot change the EDID or put any EDID in the device. Actually you cannot even use older version or newer version (newer than the FX3 version). You must use the EDID file included in the FX3 software package otherwise the EDID might be invalid and then audio source and video source will reject the dongle and not output any stream to it or use incorrect configuration (such as an unsupported color space).

Before Starting the update of the dongle EEPROM make sure that the dongle is not streaming video or audio. The new EDID for the dongle is provided by INOGENI as a .bin binary file. The EDID is located in the sector 15 (at address 0xFE00) for Spartan-6 devices and sector 63 (at address 0x3FE00) for Spartan-7 devices of the EEPROM. The sector must be erased before any data can be written in the EDID section of the EEPROM. You can erase one sector in the EEPROM using the command found in Table 14: SPI Erase Sector Command.

Example: Erase the sector 15 for Spartan-6 devices:

```
TX: 10 04 02 00 0F 00 ... 00 (total 64 bytes)
```

Command ID = 0x10 (any value from 0x01 to 0xFF)
 Erase sector command = 0x04
 Len = 0x02
 SPI dev = 0x00 (always 0x00 for the firmware update)
 Sector = 0x0F = Erase sector 15 (last sector of EEPROM)

Then wait for an answer.

```
RX: 10 04 01 00 ... 00
```

Command ID = 0x10 same than our request, that tell us this an answer to our request.
 Erase sector command = 0x04, so this answer to an erase sector.
 Len = 0x01 so will follow: 1 result code
 Result Code =0x00 this an ACK

You must wait 3 seconds after each erase command to let the SPI EEPROM time to execute the command.

After you successfully erase the sector, you can start programming the EDID in the EEPROM from the data of the .BIN file. The write of data to EEPROM is done using the command of Table 12: SPI Write Command.

Example: Write SPI EEPROM EDID at address 0xFE00 for Spartan-6 devices:

```
TX: 11 02 3C 00 0F EF 00 XX ... YY (total 64 bytes)
```

Command ID = 0x11 (any value from 0x01 to 0xFF)
 Write to EEPROM command = 0x02
 Len = 0x3C
 SPI dev = 0x00 (always 0x00 for the firmware update)
 Address [23:16] (MSB) = 0x0F
 Address [15:8] = 0xEF
 Address [7:0] (LSB) = 0x00
 Address = 0xFE00
 XX ... YY = Data 56 bytes

Then wait for an answer.

```
RX: 11 02 01 00 ... 00
```

Command ID = 0x11 same than our request, that tell us this an answer to our request.

Write to EEPROM command = 0x02, so this an answer to a write SPI

Len = 0x01 so will follow: 1 result code

Result Code =0x00 this an ACK

So you need to program the whole 256 bytes of the EDID file data into the EEPROM starting at address 0xFE00 for Spartan-6 and 0x3FE00 for Spartan-7 until the end of the file. You should always use to send write command of the maximum size of 56 bytes, except for the last write command. The figure below illustrates the flow chart for the EDID update.

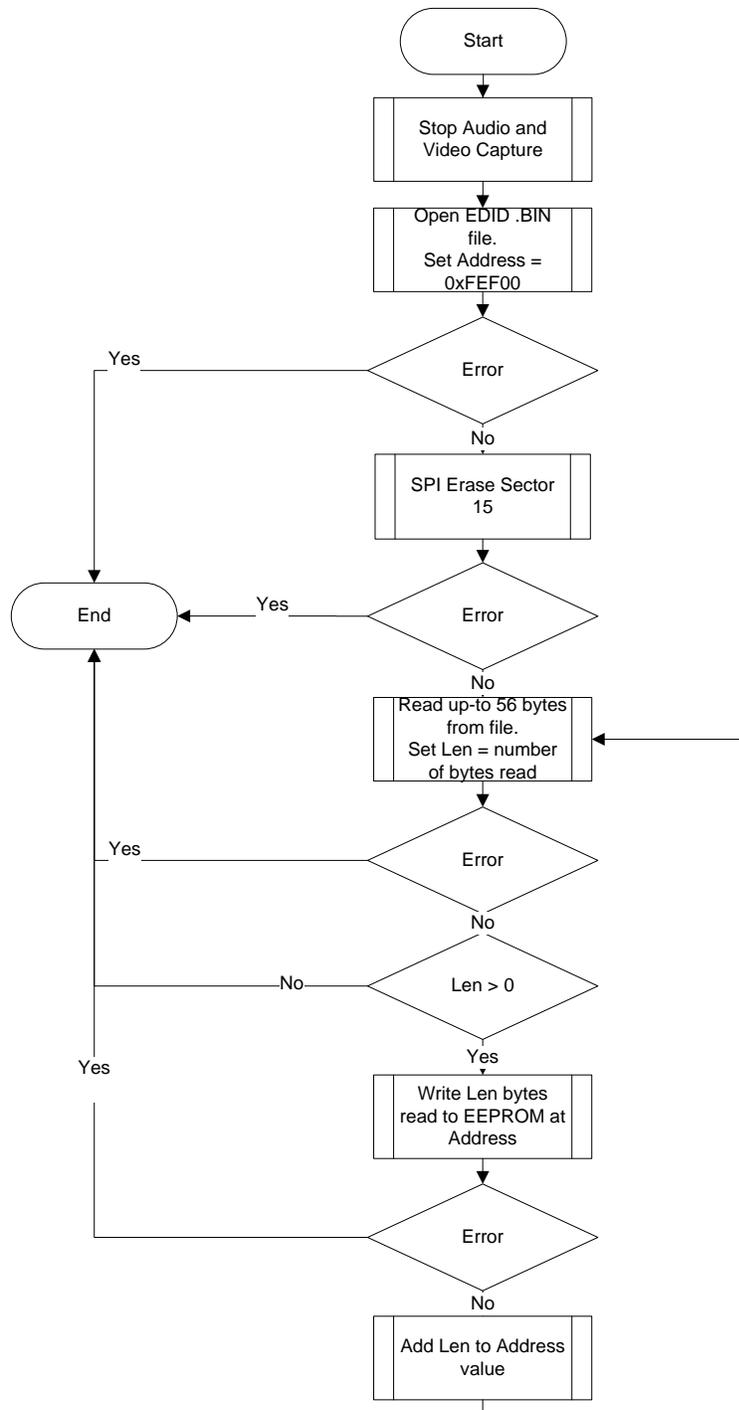


Figure 7 - EDID Update Flow

To complete the EDID update a reboot will be necessary. However, before performing a reboot you should complete the update of all part of the dongle first.

FX3 Validation

An optional step is to read back all data from the EEPROM and to compare it to the data from the .img file. This step insure that no error occur in the EEPROM update. The principle is simple: read all data in the FX3 reserved sectors and compare it to the FX3 .img file.

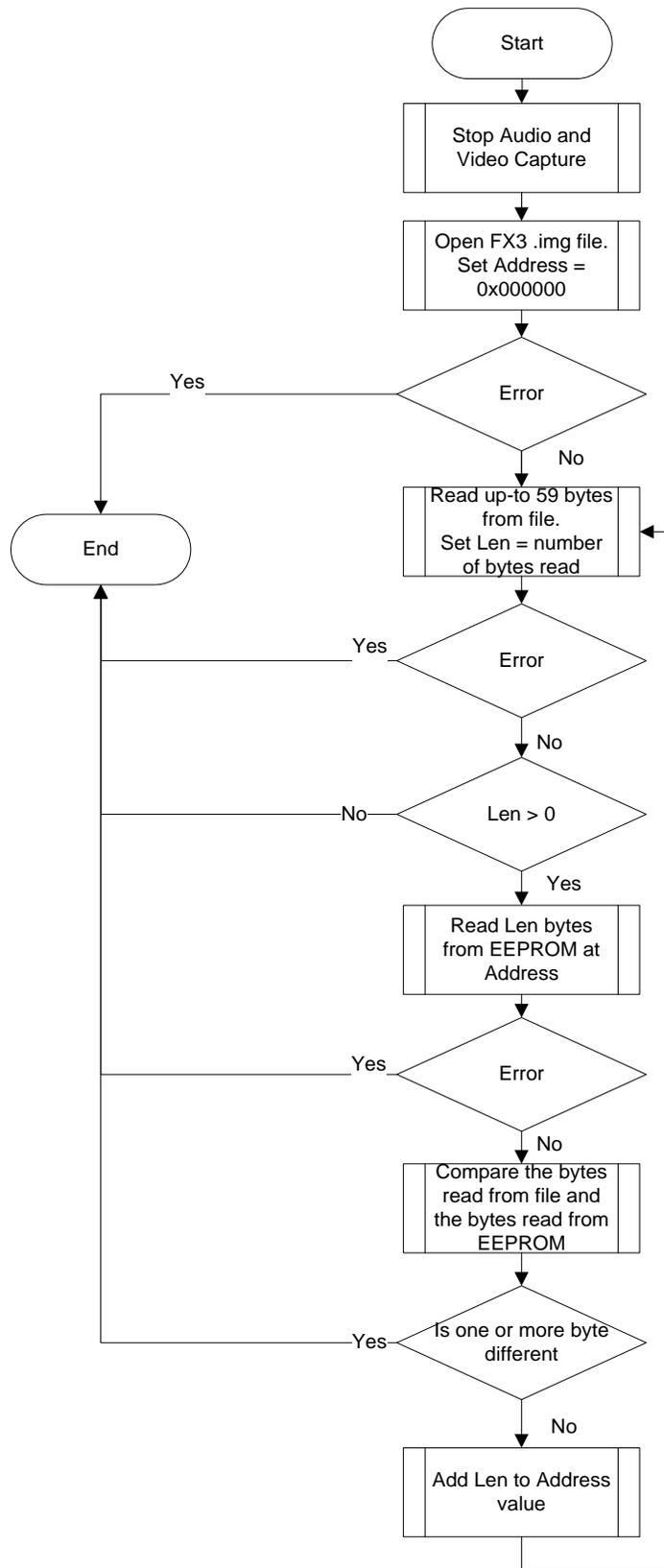


Figure 8 – FX3 Validation Flow

You simply start at address 0x00000 and verify each byte in the EEPROM is the same than the byte in the file at the corresponding offset. If any byte is different the validation has failed and the FX3 need to be updated from the start immediately. If the FX3 reboots with a corrupted EEPROM it will boot in bootloader mode (see section on Bootloader to see how to handle this case).

The only HID command needed is the read SPI (see Table 13: SPI Read Command):

Example: Read 59 byte at address 0x10100:

`TX: 11 03 05 00 01 01 00 3B ... YY (total 64 bytes)`

Command ID = 0x11 (any value from 0x01 to 0xFF)
 Read from EEPROM command = 0x03
 Len = 0x05
 SPI dev = 0x00 (always 0x00 for the firmware update)
 Address [23:16] (MSB) = 0x01
 Address [15:8] = 0x01
 Address [7:0] (LSB) = 0x00
 Address = 0x10100
 0x3B = size of the read

Then wait for an answer.

`RX: 11 03 3C 00 XX ... YY`

Command ID = 0x11 same than our request, that tell us this an answer to our request.
 Read from EEPROM command = 0x03, so this an answer to a read SPI
 Len = 0x3C so 60 bytes will follow: 1 result code + 59 data
 Result Code =0x00 this an ACK
 The data following the ACK = XX...YY

Then compare those bytes read (XX...YY) from EEPROM to the file 59 bytes starting at offset 0x10100, if one byte is different the update of the FX3 firmware has failed!

FPGA Validation

The same validation can be done with the FPGA section inside the EEPROM. This step insure that no error occur in the FPGA EEPROM update. The principle is simple: read all data in the FPGA reserved sectors and compare it to the FPGA bit file.

The size of the FPGA has been written at FPGA starting address and it should be verified first. Then you start at address `FPGA_START_ADDRESS + 0x100` and verify each byte until the end of the file. If any byte read from the EEPROM is not equal to the corresponding byte in the file, then the updated has failed and the FPGA need to be updated from the start. That should never happen; if the HID device acknowledges a write SPI command it assumes that the write has work. The chance of data being incorrectly transfer by HID is small if existent at all. This is an additional safety measure.

The SPI read command can be used to get the 5 first byte of address `FPGA_START_ADDRESS` (see Table 13: SPI Read Command):

`TX: 12 03 05 00 03 00 00 05 ... YY (total 64 bytes)`

Command ID = 0x12 (any value from 0x01 to 0xFF)
 Read from EEPROM command = 0x03
 Len = 0x05
 SPI dev = 0x00 (always 0x00 for the firmware update)
 Address [23:16] (MSB) = 0x03
 Address [15:8] = 0x00
 Address [7:0] (LSB) = 0x00
 Address = 0x30000
 5 = size of the read

Then wait for an answer.

RX: 12 03 06 00 AA BB CC DD EE 00 00 ...

Command ID = 0x12 same than our request, that tell us this an answer to our request.
 Read from EEPROM command = 0x03, so this an answer to a read SPI
 Len = 0x06 so 6 bytes will follow: 1 result code + 5 data
 Result Code = 0x00 this an ACK
 The data following the ACK = AA BB CC DD EE 00 00 ...
 buffer[0] = AA;
 buffer[1] = BB;
 buffer[2] = CC;
 buffer[3] = DD;
 buffer[4] = EE;

Verify that byte in buffer [4] is equal to the checksum of the 4 first bytes at address FPGA_START_ADDRESS:

```

for (i = 0; i < 4; i++)
{
    checksum += buffer[i];
}
checksum = ((256 - checksum) % 256);

if (buffer[4] != checksum)
{
    return false;
}
  
```

The total length retrieved from the EEPROM must match the bit file size:

```

TotalLen = (((uint32)buffer[0]) | ((uint32)buffer[1] << 8) |
((uint32)buffer[2] << 16) | ((uint32)buffer[3] << 24) );
  
```

If the validation of the size failed, there is no need to push the validation further as this FPGA is incorrectly updated. On success you need to read back the whole file and the corresponding section in the EEPROM and to compare both for any difference. The following figure is the flow chart for validating the whole FPGA bit stream.

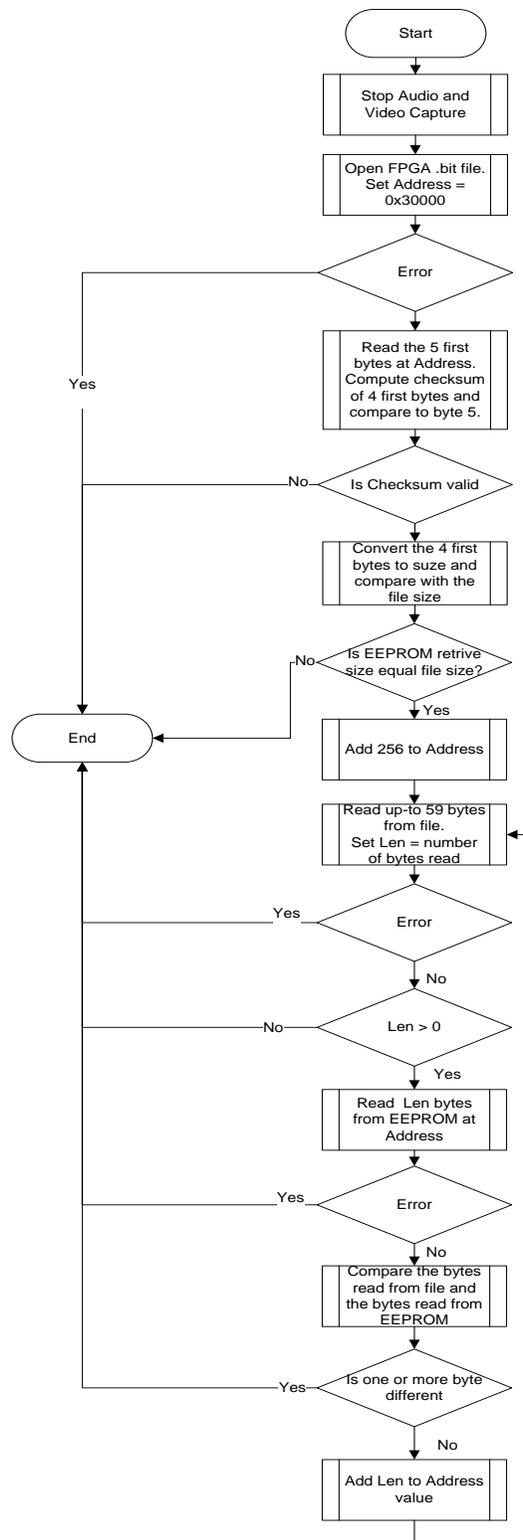


Figure 9 - FPGA Validation Flow

The only HID command needed in the validation is the SPI read (see Table 13: SPI Read Command):

Example: Read the first 59 bytes of the FPGA bit stream inside the EEPROM:

TX: 13 03 05 03 01 01 00 3B ... YY (total 64 bytes)

Command ID = 0x13 (any value from 0x01 to 0xFF)
Read from EEPROM command = 0x03
Len = 0x05
SPI dev = 0x00 (always 0x00 for the firmware update)
Address [23:16] (MSB) = 0x03
Address [15:8] = 0x01
Address [7:0] (LSB) = 0x00
Address = 0x30100
0x3B = size of the read

Then wait for an answer.

RX: 13 03 3C 00 XX ... YY

Command ID = 0x13 same than our request, that tell us this an answer to our request.
Read from EEPROM command = 0x03, so this an answer to a read SPI
Len = 0x3C so 60 bytes will follow: 1 result code + 59 data
Result Code = 0x00 this an ACK
The data following the ACK = XX...YY

Then compare those bytes read (XX...YY) from EEPROM to the file 59 first bytes, if one byte is different the update of the FPGA has failed!

If the dongle reboot before you have a chance to perform the update, the dongle will not be able to stream video or audio. In that case, simply re-start the update of the FPGA from the beginning.

EDID Validation

The EDID can also be validated after update. This step insures that no error has occurred in the EEPROM update. The principle is simple: read all data in the EDID reserved sectors and compare it to the EDID BIN file.

You simply start at address 0xFE00 for Spartan-6 and 0x3FE00 for Spartan-7 and verify each byte until the end of the file. The following figure is the EDID validation flow chart.

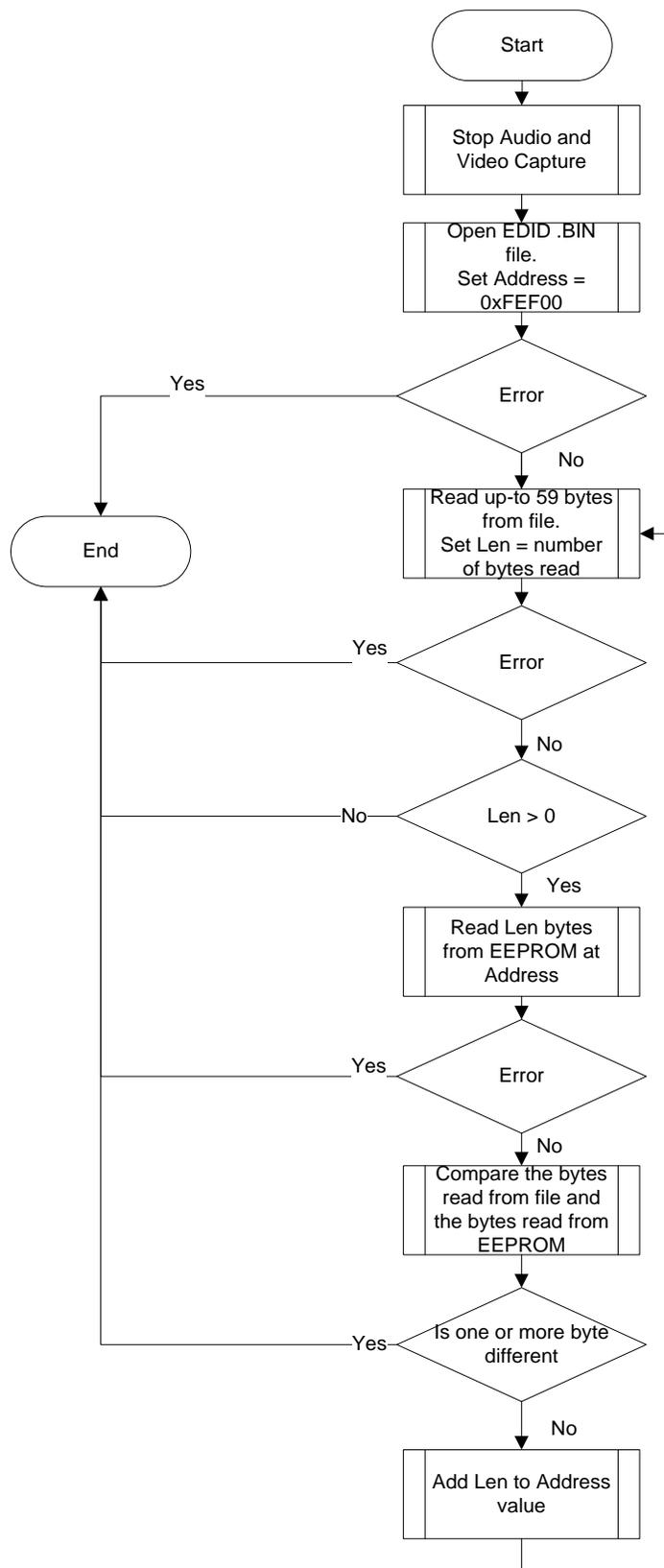


Figure 10 - EDID Validation Flow

The only HID command needed in the validation is the read SPI (see Table 13: SPI Read Command):

Example: Read the first 59 bytes of the EDID in the EEPROM:

TX: 14 03 05 00 0F FE 00 3B ... YY (total 64 bytes)

Command ID = 0x14 (any value from 0x01 to 0xFF)
 Read from EEPROM command = 0x03
 Len = 0x05
 SPI dev = 0x00 (always 0x00 for the firmware update)
 Address [23:16] (MSB) = 0x0F
 Address [15:8] = 0xFE
 Address [7:0] (LSB) = 0x00
 Address = 0xFE00
 0x3B = size of the read

Then wait for the answer.

RX: 14 03 3C 00 XX ... YY

Command ID = 0x14 same than our request, that tell us this an answer to our request.
 Read from EEPROM command = 0x03, so this an answer to a read SPI
 Len = 0x3C so 60 bytes will follow: 1 result code + 59 data
 Result Code = 0x00 this an ACK
 The data following the ACK = XX...YY

Then compare those bytes read (XX...YY) from EEPROM to the file 59 first bytes, if one byte is different the update of the EDID has failed!

If any byte is different the validation has failed. If the device is rebooted with an invalid EDID it may be impossible to receive video and audio from some sources if not from all sources. In any case of error of the validation (HID Command failed) re-start the validation from the beginning. If one byte or more read from the EEPROM is different than the byte read from the file update the EDID again from the beginning.

Final Step

After the update of the EEPROM, the dongle need to reboot. The reboot is not only necessary so the dongle use the new code but it is also necessary because we shut down some important features during the update and there is no easy and safe way of turning those features back on. The FX3 can be rebooted programmatically using the HID interface. To reboot the FX3 use the following HID command:

TX: EE 08 02 FF 01 00... 00 (total 64 bytes)

Command ID = 0xEE (any value from 0x01 to 0xFF)
 Debug Write Command = 0x08
 Len = 0x02
 Command Id = 0xFF
 Command data = 0x01

There is no answer form the dongle to that command: do not wait an ACK, however it could be rejected!

As soon as the FX3 received the command it starts to reboot. The dongle will not answer that request but you should see the device disappear from the Windows event. Allow several seconds for the dongle to reboot, even after the HID device is back wait about 7 seconds before communicating with it. Then the update should be complete and the dongle is now updated.

As an optional final validation, the version of all devices can be retrieved using HDI command to verify the version is the expected one (see section Devices Version).

11. Bootloader

There is a danger when updating the dongle FX3, if it fails and the device is power cycled, the device won't reboot. The FX3 dongle will not boot normally, it will boot in bootloader mode. In that case the HID device, the video device and audio device does not appear in device manager.

This should never happen if the updater follows the procedure, but accident does happen such as USB3 device removal or computer power failure during the firmware update.

To handle device in bootloader mode you need to import the Cypress DLL in your code, which means that the bootloader recovery is done only under Windows systems for the moment.

In the software initialisation, create the code to look for the dongle in boot mode:

```
USBDeviceList usbDevices;

CyConst.SetClassGuid("{CDBF8987-75F1-468e-8217-97197F88F773}");

usbDevices = new USBDeviceList(CyConst.DEVICES_CYUSB);

// optionally you can put an handler to detect the addition or removal of a device in bootmode
usbDevices.DeviceRemoved += new EventHandler(usbDevices_DeviceRemoved);
usbDevices.DeviceAttached += new EventHandler(usbDevices_DeviceAttached);
```

In the upload function you can now search for dongle in boot mode and update it with a temporary image file. After simply follow the normal EEPROM update for the FX3 (see FX3 Firmware Update) and after the update of the other device if necessary.

```
//Then check for a device in boot mode
foreach (USBDevice FxDev in usbDevices)
{
    //dev is a USBDevice, so a Cypress bootloader device
    // check for bootloader first, if it is not running then prompt message to user.
    if (!fx.IsBootloaderRunning())
    {
        MessageBox.Show("Please reset your device to download firmware", "Bootloader is not running");
        return;
    }

    if (FxDev != null)
    {
        FX3_FWDNLOAD_ERROR_CODE enmResult = FX3_FWDNLOAD_ERROR_CODE.SUCCESS;

        StatLabel.Text = "Programming RAM of " + FxDev.FriendlyName;
        Refresh();

        enmResult = fx.DownloadFw(filename, FX3_FWDNLOAD_MEDIA_TYPE.RAM);

        StatLabel.Text = "Programming " + fx.GetFwErrorString(enmResult);
        Refresh();
    }
}
```

12.Troubleshooting

The previous sections does cover munch what to do in case of error.

HID Command Fail

The dongle is multithreaded application that handles different module and IC at the same times. It is possible to get a BUSY reject code, especially when it is occupied at streaming video and/or audio. The result code BUSY (see Table 8: Error Code) should not be consider as an error and the software should be ready to handle them with retries. Other error code should not normally happen; if it does the software should consider the operation has failed.

Update Failure

If the device update failed due to HID command rejected or to any other error the software updater should start the update from the start. The updater should not reboot after a failed update it should try again the update, starting with the sector erase.

Update Failure FX3

The FX3 is the processor controlling the HID, if the update is incorrectly done and the device is rebooted or if the device is power cycled or remove from USB3 port during the update. The device will fall in bootloader mode, so no HID device will be visible on the computer. The device bootloader can be updated with a temporary .img file using the Cypress bootloader dll. For more information on handling a device in bootmode see the Bootloader section.

Update Failure FPGA

If the update of the FPGA failed and the device is rebooted the FPGA will not be loaded. When you send get version command the FX3 will return a 0 as FGPA version. Note that the FPGA is responsible of video and audio streaming, therefore you will not be able to do those actions if the device has rebooted with a corrupted FPGA EEPROM. However the HID device will be unaffected so simply do the update again and reboot the device.

Update Failure EDID

If the update of the EDID failed and the device is rebooted the EDID will be invalid and then rejected by all video and audio source. With an invalid EDID, audio and video source may refuse to stream video to the dongle or use unsupported settings making the audio and video to be incorrect. The HID device will be unaffected by booting with an invalid EDID, so simply do the update again. Note that checking the EDID version is not a good validation as this is only looking at a specific byte and does not do a checksum validation.

Validation Failure

If the validation of one device failed, the update should be done again before rebooting the device.

The Device Disappear

If the update of the FX3 is incorrectly done, the video device, the audio device and the HID device will not appear in the Windows Device Manager. In that case the device will appear with a different

vendor id and product id as a Cypress Boot loader device. A temporary .img file can be program using the Cypress bootloader. But note that only temporary image file is uploaded using the bootloader. Once the temporary FX3 firmware is uploaded the HID device will appear then you must upload a new FX3 firmware in the EEPROM (see FX3 Firmware Update). Note the Cypress bootloader offer a SPI firmware update but DO NOT USE IT as it will erase all other device than the FX3 and additional important data in the EEPROM.