



## DfuSe Application Programming Guide

---

### **Introduction**

This manual describes the different development stages of a Device firmware upgrade application, based on the DFU solution provided by STMicroelectronics (DfuSe).

# Contents

<b>1</b>	<b>Device Firmware Upgrade STMicroelectronics Extension</b>	<b>4</b>
<b>2</b>	<b>Pre-development phase</b>	<b>5</b>
2.1	Gathering needed files	5
2.2	STTube Driver	5
2.3	DfuSe programming interface files	5
2.4	Driver installation	5
2.4.1	Licence key	5
2.4.2	Interface GUID	5
2.4.3	Create an Inf file	6
<b>3</b>	<b>Development phase</b>	<b>7</b>
3.1	DFU Enumeration	7
3.1.1	Get device information set	7
3.1.2	Get number of elements in the device information set	7
3.1.3	Get details about a device interface	7
3.1.4	Open the DFU driver	9
3.1.5	Get device descriptor	9
3.1.6	Get DFU Descriptor	9
3.2	DFU Identification	9
3.2.1	Check for DFU protocol version	9
3.2.2	Check for firmware operating mode	10
3.2.3	Check for supported DFU operations	10
3.3	DFU Operations	11
3.3.1	General DFU operation	11
3.3.2	DETACH operation	11
3.3.3	RETURN operation	12
3.3.4	UPLOAD operation	12
3.3.5	ERASE operation	14
3.3.6	UPGRADE operation	14
3.4	DFU basic requests	16
3.4.1	DFU_DNLOAD	16
3.4.2	DFU_UPLOAD	16
3.4.3	DFU_GETSTATUS	17

---

3.4.4	DFU_GETSTATE .....	17
3.4.5	DFU_ABORT .....	18
3.5	Managing DFU Image .....	18
3.5.1	Get Image settings .....	18
3.5.2	Create a DFU image .....	18
3.5.3	Add an image element .....	19
3.5.4	Remove an image element .....	20
3.5.5	Store a DFU Image .....	20
3.5.6	Load a DFU Image .....	20
<b>4</b>	<b>Document references .....</b>	<b>21</b>
<b>5</b>	<b>Revision history .....</b>	<b>22</b>

# 1      **Device Firmware Upgrade STMicroelectronics Extension**

The reason for creating this DFU extension is that the standard device firmware upgrade protocol is too specialized in terms of protocol versus the target or the application.

This extension makes it easy to use DFU with all 8 or 32-bit microcontrollers, simply letting the PC side know the target memory mapping. In addition, the DFU file format has been updated, to be able to build DFU files from standard 8 or 32-bit s19, hex or bin formats. The result is a new revision of the standard DFU revision 1.1, called DfuSe for Device firmware upgrade STMicroelectronics Extension.

## 2 Pre-development phase

### 2.1 Gathering needed files

You will need to gather the needed files before starting application development, such as driver and DfuSe programming interface files:

### 2.2 STTube Driver

- The correct inf file for the operating system that you are using.
- STTub203.sys
- STTubeDevice203.dll

The sys file and the inf file must be put in the same location on the installation medium.

### 2.3 DfuSe programming interface files

- STDFU.dll
- STDFUFiles.dll
- STDFUPRT.dll

These files are used as a programming interface for DFU applications.

### 2.4 Driver installation

#### 2.4.1 Licence key

To prevent software piracy, STMicroelectronics has implemented a licence key mechanism for the Tube driver. This mechanism is based on the Vendor ID and the Product ID(s), extracted from the device(s). Contact your nearest sales office and give your Vendor ID and Product ID(s). In return you will get licence key(s), made up of 16 letters.

#### 2.4.2 Interface GUID

If you, as a programmer, want to write an application working with the driver, you will have to program a dialog box user interface, in which the user is asked to choose the device he wants to work with. However the Tube driver can be used for many purposes. For example, STMicroelectronics uses this package internally for product development. So you will not want the dialog box to display all the devices that using the Tube driver; there could be too many; and it could be sometimes meaningless to use them with a given application. This is where the concept of interface GUIDs comes in. With this mechanism, the driver is able to propose multiple access interfaces. This can be considered as an auto-declaration of the available entry points. An application will be able then to display only devices for which a given interface is available. GUID stands for “Globally Unique Identifier”, and is a Microsoft notion.

In our case, we define two GUIDs, for DFU and Application modes

GUID\_DFU = {3FE809AB-FB91-4CB5-A643-69670D52366E}

GUID\_APP = {CB979912-5029-420a-AEB1-34FC0A7D5726}

Otherwise, GUIDs can be redefined according to the developer's choice. To generate new GUIDs use the GUIDgen Microsoft tool contained in this package.

### 2.4.3 Create an Inf file

Now you have your licence keys and your GUIDs for all your products, you need to make your own inf file. The inf file is a text file used by Windows to install a driver. The one given in this package can be customized to fit your needs. To take an example, let's imagine you want to install the driver for two products. So you have two licence keys:

Product 1: 0123456789ABCDEF

Product 2: 02468ACE13579BDF

In the 1st product, you have the necessary firmware for, let's say, a joystick. The joystick function will then be the first one using the Tube Driver so you will need a GUID for it. In the 2nd product, you have a firmware with the Joystick function, and a Communication function. So you need a second GUID for this product:

Guid1 (For Joystick usage): {E91512C1-E5A3-11d5-971E-0050041B1E9F}

Guid2 (For Communication Usage): {F4CCF380-E5A3-11d5-971E-0050041B1E9F}

So an application wanting to display all devices that use the Joystick will search for all devices with the Guid1.

Now you need to change the inf file:

1. Locate the section [STTub203.AddLicences]
2. Add the following lines in this section:

```
; Product 1 Key and interfaces
HKR,0123456789ABCDEF,"{E91512C1-E5A3-11d5-971E-0050041B1E9F}",, " "
; Product 2 Key and interfaces
HKR,02468ACE13579BDF,"{E91512C1-E5A3-11d5-971E-0050041B1E9F}",, " "
HKR,02468ACE13579BDF,"{F4CCF380-E5A3-11d5-971E-0050041B1E9F}",, " "
```

Then with this customized inf file, the correct values will be entered in the registry while the driver is installed.

## 3 Development phase

### 3.1 DFU Enumeration

This topic illustrates how user-mode software can enumerate DFU device interfaces and obtain a USB Device handle. The process consists of six steps:

#### 3.1.1 Get device information set

The user-mode software calls SetupDiGetClassDevs to query for information about all of the registered device interfaces in the device interface class that is associated with the given GUID. Then it returns a handle to a device information set that contains information about the device interfaces.

```
// GUID for DFU mode
static GUID GUID_DFU = { 0x3fe809ab, 0xfb91, 0x4cb5, {
0xa6, 0x43, 0x69, 0x67, 0x0d, 0x52, 0x36, 0x6e } };
// GUID for application mode
static GUID GUID_APP = { 0xcb979912, 0x5029, 0x420a, { 0xae, 0xb1,
0x34, 0xfc, 0xa, 0x7d, 0x57, 0x26 } };

GUID Guid = GUID_DFU; // Set DFU mode GUID;
HDEVINFO info; // Device information set
```

This gets a handle to a device information set that contains all installed devices that matched the supplied parameters.

```
Info = SetupDiGetClassDevs(&Guid, NULL, NULL, DIGCF_PRESENT |
DIGCF_INTERFACEDevice);
```

If the operation fails, the function returns INVALID\_HANDLE\_VALUE or another appropriate error.

#### 3.1.2 Get number of elements in the device information set

The user-mode software must iteratively call SetupDiEnumDeviceInterfaces to determine how many elements are in the device information set.

```
SP_DEVICE_INTERFACE_DATA DeviceData;
DeviceData.cbSize = sizeof (SP_INTERFACE_DEVICE_DATA);
for (int Index = 0; SetupDiEnumDeviceInterfaces (info, NULL, & Guid,
Index, &DeviceData); Index++)
{
    ...
}
```

When there are no more interfaces, SetupDiEnumDeviceInterfaces function fails and returns false, and GetLastError returns ERROR\_NO\_MORE\_ITEMS.

### 3.1.3 Get details about a device interface

The user-mode software must enumerate the registered device interfaces that are associated with the device interface class. The application calls `SetupDiEnumDeviceInterfaces` iteratively, once again; but now it retrieves hardware identifiers (IDs) and symbolic links for each registered interface. The application uses the symbolic links to obtain a device interface handle.

```
DWORD RequiredSize;//required size of the DeviceInterfaceDetailData
//buffer
```

Getting details about device interface is typically a two-step process:

1. Get the required buffer size. Call `SetupDiGetDeviceInterfaceDetail` with a NULL `DeviceInterfaceDetailData` pointer, a `DeviceInterfaceDetailDataSize` of zero, and a valid `RequiredSize` variable. In response to this call, this function returns the required buffer size at `RequiredSize` and fails with `GetLastError` returning `ERROR_INSUFFICIENT_BUFFER`.

```
SetupDiGetDeviceInterfaceDetail (info, & DeviceData, NULL, 0,
&RequiredSize, NULL);
```

2. Allocate an appropriately sized buffer and call the function again to get the interface details. The interface detail returned by this function consists of a device path that can be passed to Win32 functions such as `CreateFile`.

```
PSP_INTERFACE_DEVICE_DETAIL_DATA detail;
detail=(PSP_INTERFACE_DEVICE_DETAIL_DATA) new BYTE[RequiredSize];
detail->cbSize=sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
SP_DEVINFO_DATA dev_info_data= {sizeof (SP_DEVINFO_DATA)};
if(SetupDiGetDeviceInterfaceDetail(info, &ifData, detail,
RequiredSize, NULL, &dev_info_data))
printf(detail->DevicePath);//DevicePath is the symbolic link of the
//DFU device.
```

Retrieve a specified Plug and Play device property, if `SPDRP_DEVICEDESC` value is set, retrieved property is a `REG_SZ` string containing the description of a device.

```
charProduct[253];//buffer, the device description will be copied to.
if(SetupDiGetDeviceRegistryProperty (info, &dev_info_data,
SPDRP_DEVICEDESC, NULL, (PBYTE)Product, 253, NULL))
printf(" -> %s\n", Product); // print the name of the device
delete[] (PBYTE) detail;
```



### 3.1.4 Open the DFU driver

The user-mode software must use the STDFU\_Open API imported from STDFU library; the szDevicePath argument is the symbolic link of the target device.

```
HANDLE hDle;  
if(STDFU_Open((LPSTR) detail->DevicePath, &hDle)==STDFU_NOERROR)  
{ STDFU_Close(&hDle);}
```

### 3.1.5 Get device descriptor

The user-mode software must use the STDFU\_GetDeviceDescriptor API imported from STDFU library, the hDle variable is the device handle pointer returned by the STDFU\_Open API.

```
HANDLE hDle;  
USB_DEVICE_DESCRIPTOR DeviceDesc;  
if (STDFU_GetDeviceDescriptor(&hDle, &DeviceDesc) == STDFU_NOERROR)  
{  
    CString Tmp;  
    Tmp.Format("%04X", DeviceDesc.idVendor) ; Printf(Tmp);  
    Tmp.Format("%04X", DeviceDesc.idProduct); Printf(Tmp);  
    Tmp.Format("%04X", DeviceDesc.bcdDevice); Printf(Tmp);  
}
```

### 3.1.6 Get DFU Descriptor

The user-mode software must use the STDFU\_GetDFUDescriptor API imported from STDFU library, the hDle variable is the device handle pointer returned by the STDFU\_Open API. The bmAttributes field provides information about the available DFU requests and DFU protocol version.

```
HANDLE hDle;  
DFU_FUNCTIONAL_DESCRIPTOR DFUDesc;  
UINT DFUInterfaceNum;  
UINT NbofAlternates;  
memset(&DFUDesc, 0, sizeof(DFUDesc));  
if(STDFU_GetDFUDescriptor(&hDle,&DFUInterfaceNum,&NbofAlternates,  
&DFUDesc)==STDFUPRT_NOERROR)  
{ ... }
```

## 3.2 DFU Identification

### 3.2.1 Check for DFU protocol version

The DFU firmware version is coded in two bytes, which can be read from the bcdDFUVersion attribute in the retrieved DFU descriptor returned by the STDFU\_GetDFUDescriptor call in DFU mode.

```
if((DFUDesc.bcdDFUVersion<0x011A)|| (DFUDesc.bcdDFUVersion>=0x0120))  
    printf("Bad DFU protocol version. Should be 1.1A");
```

### 3.2.2 Check for firmware operating mode

To apply DFU operations, the firmware must be run in DFU mode. Consequently the user-mode software should verify if the device is in DFU mode by comparing the current GUID to the default DFU GUID.

```
// Tries to know if we are in DFU or in Application mode: based on  
the GUID  
CString DevGUID= DFUName.Right(38);  
CString DfuGUID= "{3FE809AB-FB91-4CB5-A643-69670D52366E}";  
if(DevGUID.CompareNoCase(DfuGUID)!=0)  
{  
    // Device in Application Mode  
}  
else  
{  
    // Device in DFU Mode  
}
```

### 3.2.3 Check for supported DFU operations

When device enumeration is finished and before launching a DFU operation, you should check for supported operations using the `bmAttributes` attribute in the DFU descriptor record.

```
DFU_FUNCTIONAL_DESCRIPTOR DFUDesc;  
UINT D1, D2;  
memset(&m_CurrDevDFUDesc, 0, sizeof(DFUDesc));  
  
STDFU_GetDFUDescriptor(&hDle, &D1, &D2, &DFUDesc);  
  
if(DFUDesc.bmAttributes & ATTR_DNLOAD_CAPABLE)  
    printf (CanDnload);  
if(DFUDesc.bmAttributes & ATTR_UPLOAD_CAPABLE)  
    printf (CanUpload);  
if(DFUDesc.bmAttributes & ATTR_WILL_DETACH)  
    printf (CanDetach);  
if(DFUDesc.bmAttributes & ATTR_MANIFESTATION_TOLERANT)  
    printf(CanManifestTolerant);  
if(DFUDesc.bmAttributes & ATTR_ST_CAN_ACCELERATE)  
    printf (CanAccel);
```

## 3.3 DFU Operations

### 3.3.1 General DFU operation

Use the `DFUThreadContext` class type to program a DFU operation, by setting the appropriate values for the available parameters according to the requested operation. For all requests, the `szDevLink`, `DfuGUID`, `AppGUID` attributes must be fixed according to the selected DFU device.

```
DWORD OperationCode; // reference to the launched operation.  
...  
DFUThreadContext Context;  
lstrcpy (Context.szDevLink, DFU_Name);  
Context.DfuGUID=GUID_DFU;  
Context.AppGUID=GUID_APP;
```

The `Operation` attribute is used to identify the operation to be launched; it contains one of the following codes:

(`OPERATION_DETACH`, `OPERATION_RETURN`, `OPERATION_UPLOAD`,  
`OPERATION_ERASE` and `OPERATION_UPGRADE`).

```
Context.Operation = OPERATION_UPGRADE;
```

The `bDontSendFFTransfersForUpgrade` attribute is used only for download operations; it means that the FFh bytes in the data to be downloaded will not be sent.

```
Context.bDontSendFFTransfersForUpgrade = TRUE;
```

The `hImage` attribute is a handle image pointer, used when a download, erase or upload operation is to be launched.

```
Context.hImage=hImage;
```

The `OperationCode` argument contains a reference to the launched operation after calling `STDFUPRT_LaunchOperation` API.

```
dwRet = STDFUPRT_LaunchOperation(&Context, &OperationCode);
```

### 3.3.2 DETACH operation

The detach operation is used during the reconfiguration phase to apply the selected configuration.

```
DWORD OperationCode;
```

```

DFUThreadContext Context;
DWORD dwRet;
HANDLE hImage;
lstrcpy(Context.szDevLink, DFUName); // DFUName is the device
symbolic link
Context.DfuGUID=GUID_DFU;
Context.AppGUID=GUID_APP;
Context.Operation=OPERATION_DETACH;
Context.hImage=NULL;
dwRet=STDFUPRT_LaunchOperation(&Context, &OperationCode);
if (dwRet != STDFUPRT_NOERROR)
Context.ErrorCode=dwRet;

```

### 3.3.3 RETURN operation

To leave FDU mode and return to Application mode, use the return operation.

```

DWORD OperationCode;
DFUThreadContext Context;
DWORD dwRet;
HANDLE hImage;
Int TargetSel = 0;
CString Name;

lstrcpy(Context.szDevLink, DFUName);
Context.DfuGUID=GUID_DFU;
Context.AppGUID=GUID_APP;
Context.Operation=OPERATION_RETURN;
(STDFUPRT_CreateMappingFromDevice((LPSTR) (LPCSTR)DFUName,
&pMapping, &NbAlternates);
Name = pMapping[i].Name;
STDFUFILES_CreateImageFromMapping(&hImage, pMapping+TargetSel);
STDFUFILES_SetImageName(hImage, (LPSTR) (LPCSTR)Name);
STDFUFILES_FilterImageForOperation(hImage, pMapping + TargetSel,
OPERATION_RETURN, FALSE);
Context.hImage=hImage;
dwRet=STDFUPRT_LaunchOperation(&Context, &OperationCode);
if (dwRet!=STDFUPRT_NOERROR)
    Context.ErrorCode=dwRet;

```

### 3.3.4 UPLOAD operation

To launch an upload operation, the user-mode software starts by retrieving details about the device mapping using a STDFUPRT\_CreateMappingFromDevice call.

```

DWORD OperationCode;
int TargetSel=0;
PMAPPING pMapping;
STDFUPRT_CreateMappingFromDevice((LPSTR) (LPCSTR)DFUName, &pMapping+
TargetSel, &NbAlternates);

```

Then the pMapping record is used to extract the DFU image from the device, a name is attributed to identify the image in the file.

```
HANDLE hImage;
CString Name = 'dfu_image_name';
STDFUFILES_CreateImageFromMapping(&hImage, pMapping+TargetSel);
STDFUFILES_SetImageName(hImage, (LPSTR) (LPCSTR)Name);
```

Before saving the image, a filter must be performed for the upload operation, using a STDFUFILES\_FilterImageForOperation call.

```
STDFUFILES_FilterImageForOperation(hImage, pMapping+TargetSel,
OPERATION_UPLOAD, FALSE);
```

When this is done, the upload operation can be launched using a DFUThreadContext class instance.

```
lstrcpy (Context.szDevLink, DFUName);
Context.DfuGUID=GUID_DFU;
Context.AppGUID=GUID_APP;
Context.Operation=OPERATION_UPLOAD;
Context.hImage=hImage;
dwRet = STDFUPRT_LaunchOperation(&Context, &OperationCode);
if (dwRet!=STDFUPRT_NOERROR) Context.ErrorCode=dwRet;
```

When the operation is successfully terminated, the image is ready to be stored in a dfu file. At this stage two cases arise, either to store the image in a new file or in an existing file.

- In the first case use the STDFUFILES\_CreateNewDFUFile method.

```
WORD Vid=0x0483;
WORD Pid=0xDF11;
WORD Bcd=0x011A;
CString UpFileName = 'new dfu file';
dwRet=STDFUFILES_CreateNewDFUFile((LPSTR) (LPCSTR)UpFileName,&hFile,
Vid, Pid,Bcd);
```

The Vid, PID and Bcd values are used to identify the image as usable for devices with given identifiers

- In the second case the STDFUFILES\_OpenExistingDFUFile method is to be used.

```
dwRet=STDFUFILES_OpenExistingDFUFile((LPSTR) (LPCSTR)UpFileName,
&hFile, NULL, NULL, NULL, NULL);
```

To store the image use STDFUFILES\_AppendImageToDFUFile API.

```
dwRet=STDFUFILES_AppendImageToDFUFile(hFile, Context.hImage);
```

To upload a personalized binary data please refer to DFU\_UPLOAD request([3.4.2](#)).

### 3.3.5 ERASE operation

The Erase operation consists of setting all non FFh bytes to FFh, this operation is indispensable for download operations. In fact the download can write only on the bits with 1. To optimize the operation time, we should create an appropriate image for the Erase operation, by creating an image from the device mapping and filtering it for the operation.

```
DWORD OperationCode;
PDWORD pNbAlternates;
PMAPPING pMapping;
PHANDLE pHandle;
// Programming the operation contex
lstrcpy(Context.szDevLink, DFUName);
Context.DfuGUID=GUID_DFU;
Context.AppGUID=GUID_APP;
Context.Operation=OPERATION_ERASE;
Context.bDontSendFFTransfersForUpgrade= TRUE;

STDFUPRT_CreateMappingFromDevice((LPSTR)(LPCSTR)DFUName,&pMapping,
&NbAlternates);
STDFUFILES_CreateImageFromMapping(pHandle, pMapping);
STDFUFILES_FilterImageForOperation(hImage, m_pMapping+TargetSel,
OPERATION_ERASE, FALSE);
Context.hImage=hImage;
if( STDFUPRT_LaunchOperation(&Context, &OperationCode) !=
STDFUPRT_NOERROR)
{
    Printf('Erase error');
}
```

**Note:** *As the download operation should be preceded by an Erase operation, the erase image can be created as a duplication of the download image using the STDFUFILES\_DuplicateImage and then filtered for Erase.*

### 3.3.6 UPGRADE operation

The Upgrade (Download) operation consists of downloading a firmware image into device Flash memory. It can be done only with a specific image structure built from a DFU file or from device mapping using STDFUFILES\_ReadImageFromDFUFile or STDFUFILES\_CreateImageFromMapping methods respectively.

To launch an Upgrade (download) operation, the user-mode software opens a compatible DFU file, by calling the STDFUFILES\_OpenExistingDFUFile method, the NbImages field counts the available images in the file.

```
DWORD OperationCode;
DWORD dwRet;
HANDLE hFile;
BYTE NbImages;
CString DownFileName = "DnLoadfilename.dfu";
dwRet=STDFUFILES_OpenExistingDFUFile( (LPSTR) (LPCSTR)DownFileName,
&hFile, NULL, NULL, NULL, &NbImages);
```

When successful, the call returns a STDFUFILES\_NOERROR value. The DFU file can contain more than one image, in this case you must retrieve the suitable image for the selected device interface, by comparing the device index to the Alternate value of all available images.

```
int TargetSel=0;
BOOL bFound;
HANDLE hImage;
for(int i=0;i<NbImages;i++)
{
    HANDLE Image;
    BYTE Alt;
    if(STDFUFILES_ReadImageFromDFUFile(hFile,i,
    &Image)==STDFUFILES_NOERROR)
    {
        if(STDFUFILES_GetImageAlternate(Image,
    &Alt)==STDFUFILES_NOERROR)
        {
            if (Alt==TargetSel)
            {
                hImage =Image;
                bFound =TRUE;
                break;
            }
            STDFUFILES_DestroyImage(&Image);
        }
    }
    STDFUFILES_CloseDFUFile(hFile);
}
```

If the suitable image is recovered, and before programming the download operation you must retrieve the mapping details from the device, and filter the image for download.

```
PMAPPING pMapping;
DWORD NbAlternates;
BOOL UpgradeOptimized = TRUE;
CString DFU_Name ; // Device symbolic link.
```

```
STDFUPRT_CreateMappingFromDevice ((LPSTR) (LPCSTR) DFUName,
&pMapping, &NbAlternates);
STDFUFILES_FilterImageForOperation(hImage, pMapping+TargetSel,
OPERATION_UPGRADE, UpgradeOptimized);
```

The UpgradeOptimized Boolean value is used to indicate if the upgrade operation ignores the FFh data

Now, one instance of DFUThreadContext class is to be created and programmed to launch the operation.

```
DFUThreadContext Context;
lstrcpy (Context.szDevLink, DFU_Name);
Context.DfuGUID=GUID_DFU;
Context.AppGUID=GUID_APP;
Context.Operation=OPERATION_UPGRADE;
Context.bDontSendFFTransfersForUpgrade= TRUE;
Context.hImage=hImage;
dwRet=STDFUPRT_LaunchOperation(&Context, &OperationCode);
if (dwRet!=STDFUPRT_NOERROR)
{
    Context.ErrorCode=dwRet;
}
```

To upgrade a personalized binary data please refer to DFU\_DNLOAD request ([Section 3.4.1](#)).

## 3.4 DFU basic requests

### 3.4.1 DFU\_DNLOAD

Download the given binary data into the device flash memory at the specified address. Use the STDFU\_Dnload to perform this request.

```
HANDLE hDle;
BYTE* pBuffer; // Data buffer to be downloaded
ULONG nBytes; // Size of data in byte
USHORT nBlock; // Block number
... // Copy data to be downloaded into pBuffer
if (STDFU_Open((LPSTR) DFUName, &hDle)==STDFU_NOERROR)
{
    STDFU_Dnload(hDle, pBuffer, nBytes, nBlock)
    STDFU_Close(&hDle);
}
```

### 3.4.2 DFU\_UPLOAD

The purpose of Upload is to provide the capability to retrieve and archive the firmware of a device. Use STDFU\_Upload to perform this request.



```

HANDLE hDle;
BYTE* pBuffer; // buffer, the uploaded data will be copied to.
ULONG nBytes; // Size of data in byte
USHORT nBlock; // Block number
...
if (STDFU_Open((LPSTR) DFUName, &hDle)==STDFU_NOERROR)
{
    STDFU_Upload(hDle, pBuffer, nBytes, nBlock)
    STDFU_Close(&hDle);
}

```

### 3.4.3 DFU\_GETSTATUS

To get the current DFU Status, use the STDFUPRT\_GetOperationStatus method. The LastDFUStatus record in the DFUThreadContext class instance given as second argument will be updated, and you can retrieve the status using the bStatus attribute.

```

CString Tmp;
DFUThreadContext Context;
DWORD OperationCode;
... // programm Context for requested operation.
STDFUPRT_LaunchOperation(&Context, OperationCode)
// The OperationCode is the operation reference returned by
//LaunchOperation call.
STDFUPRT_GetOperationStatus(OperationCode, &Context);
Tmp.Format("DFU Status: Context->LastDFUStatus.bStatus");
Printf(Tmp);

```

The user-mode software can use the basic STDFU\_GetStatus API directly:

```

PHANDLE phDevice;
DFUSTATUS * DfuStatus;
...
STDFU_Getstatus(phDevice, DfuStatus);

```

### 3.4.4 DFU\_GETSTATE

To get the current DFU State, use the STDFUPRT\_GetOperationStatus method. The LastDFUStatus record in the DFUThreadContext class instance given as second argument will be updated, and you can retrieve the status using the bState attribute.

```

CString Tmp;
DFUThreadContext Context;
DWORD OperationCode;
... // programm Context for requested operation.
STDFUPRT_LaunchOperation(&Context, OperationCode)
// The OperationCode is the operation reference returned by
//LaunchOperation call.
STDFUPRT_GetOperationStatus(OperationCode, &Context);

```

```
Tmp.Format("DFU Status: Context->LastDFUStatus.bStatus");  
Printf(Tmp);
```

The user-mode software can use the basic STDFU\_GetState API directly:

```
UCHAR* pState;  
PHANDLE phDevice;  
...  
STDFU_Getstate(phDevice, pState);
```

### 3.4.5 DFU\_ABORT

To send an abort request to the DFU device use the STDFU\_Abort method, knowing that this request is sent automatically by the operation, to do this use the STDFUPRT\_StopOperation method.

```
DFUThreadContext Context;  
PHANDLE phDevice;  
...  
// Send an Abort request to the DFU device.  
STDFU_Abort(phDevice);  
...  
// Then stop a launched operation  
// The OperationCode, retrieves after STDFUPRT_LaunchOperation call  
// references the operation to be stopped.  
STDFUPRT_StopOperation(OperationCode, &Context);
```

## 3.5 Managing DFU Image

The STDFUFiles library provides a complete programming interface for managing DFU image.

### 3.5.1 Get Image settings

Before using an image, you should get its information, such as the associated alternate, the image name and the number of elements.

To get the associated alternate use the STDFUFILES\_GetImageAlternate API.

To get the image name use the STDFUFILES\_GetImageName API.

To get the number of elements use STDFUFILES\_GetImageNbElement API.

### 3.5.2 Create a DFU image

To create a DFU image, two functions are available, you can either create a customized image according to the device mapping, or just create an empty image.

The following example describes how to create a customized image:

```
int TargetSel=0;  
BYTE NbAlternates;
```

```

PMAPPING pMapping;
HANDLE hImage;
CString Name = 'dfu_image_name';

//retrive device mapping and number of alternates for used device.
STDFUPRT_CreateMappingFromDevice((LPSTR) (LPCSTR)DFUName,&pMapping+
TargetSel, &NbAlternates);
//Create a new image according to recovered mapping
STDFUFILES_CreateImageFromMapping(&hImage, pMapping+TargetSel);
//Set image name
STDFUFILES_SetImageName(hImage, (LPSTR) (LPCSTR)Name);

```

The following example describes how to create an empty image:

```

BYTE NbAlternates;
HANDLE hImage;
CString Name = 'dfu_image_name';

//Create a new empty image
STDFUFILES_CreateImage(&hImage, NbAlternates);
//Set image name
STDFUFILES_SetImageName(hImage, (LPSTR) (LPCSTR)Name);

```

### 3.5.3 Add an image element

As an image is composed of a list of image elements which contains the actual firmware data, some functions are provided to insert, replace or remove an element.

To insert or replace an element in an image, use the STDFUFILES\_SetImageElement API as follows:

```

HANDLE hImage;
DWORD ElementIndex = 0;
CString Name = 'dfu_image_name';
DFUIMAGEELEMENT Element;
BOOL bInsert=TRUE;//Element will be inserted at the specified index.

// Create a new empty image
STDFUFILES_CreateImage(&hImage, NbAlternates);
// Set image name
STDFUFILES_SetImageName(hImage, (LPSTR) (LPCSTR)Name);

// Create an Image Element
Element.dwDataLength = ... ; // Set data size in byte.
Element.dwAddress= ...; // Set element starting address.
Element.Data=new BYTE[Element.dwDataLength]; // alloc data buffer.

// Add the element
if (STDFUFILES_SetImageElement(Image,ElementIndex,bInsert,
Element)!=STDFUFILES_NOERROR)
{

```

```
    printf("Unable to insert one element in the image...");  
}
```

To replace an existing element, set `blInsert` as `FALSE`.

### 3.5.4 Remove an image element

To remove an existing element use the `STDFUFILES_DestroyImageElement` API as follows:

```
HANDLE hImage;  
DWORD ElementIndex = ...;  
...  
STDFUFILES_DestroyImageElement(hImage, ElementIndex);
```

You can retrieve the element index using `STDFUFILES_GetImageElement` API.

### 3.5.5 Store a DFU Image

As a DFU image can be used more than one time, we need to store it in a file, again, our library provides this functionality. The `STDFUFILES_AppendImageToDFUFile` API allows you either to store the image into a new DFU file, or to added it into an existing one. Please refer to the [UPLOAD](#) operation example ([Section 3.3.4](#))

Otherwise you can save a DFU image in s19 or Hex files, to do that, use the `STDFUFILES_ImageFromFile` call.

```
STDFUFILES_ImageFromFile(PSTR pPathFile, PHANDLE pImage, BYTE  
nAlternate)
```

The file type is recognized by its extension (\*.S19 or \*.Hex).

### 3.5.6 Load a DFU Image

To retrieve a stored image from an existing DFU file use the `STDFUFILES_ReadImageFromDFUFile` API. Please refer to the [UPGRADE](#) operation example ([Section 3.3.6](#))

Otherwise you can retrieve a DFU image from s19 or Hex files, to do that, use the `STDFUFILES_ImageFromFile` call:

```
STDFUFILES_ImageFromFile(PSTR pPathFile, PHANDLE pImage, BYTE  
nAlternate)
```

The file type is recognized by its extension (\*.S19 or \*.Hex).

## 4 Document references

**Table 1. Document references**

ID	Document
UM0392	DfuSe Application Programming Interface
UM0391	DfuSe File Format Specification

## 5 Revision history

**Table 2. Document revision history**

Date	Revision	Changes
06-Jun-2007	1	Initial release.

**Please Read Carefully:**

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

**UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.**

**UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.**

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2007 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

[www.st.com](http://www.st.com)

