DynaXDF 4.0

Reference & Manual

API Reference Version 4.0.102

June 9, 2025

Legal Notices

DynaForms GmbH
Bokel 3
D-49626 Bippen, Germany
Trade Register HRB 218421, District Court Osnabrueck
CEO Jens Boschulte
Phone: ++49 5435-955 3226
Fax: ++49 5435-956 7922

If you have questions please send an email to info@dynaforms.com, or contact us by phone.

LibPNG, PNG Reference Library, Copyright 1998-2004 Glenn Randers-Pehrson.

LibTIFF, TIFF Image Library, Copyright 1988-1997 Sam Leffler, Copyright 1991-1997 Silicon Graphics, Inc.

OpenJPEG, JPEG 2000 Image Library

Copyright (c) 2002-2014, Universite catholique de Louvain (UCL), Belgium
Copyright (c) 2002-2014, Professor Benoit Macq
Copyright (c) 2001-2003, David Janssens
Copyright (c) 2002-2003, Yannick Verschueren
Copyright (c) 2003-2007, Francois-Olivier Devaux
Copyright (c) 2003-2014, Antonin Descampe
Copyright (c) 2005, Herve Drolon, FreeImage Team
Copyright (c) 2006-2007, Parvatha Elangovan
Copyright (c) 2008, Jerome Fimes, Communications & Systemes <jerome.fimes@c-s.fr>
Copyright (c) 2010-2011, Kaori Hagihara
Copyright (c) 2011-2012, Centre National d'Etudes Spatiales (CNES), France
Copyright (c) 2012, CS Systemes d'Information, France

Zint barcode library, Copyright 2008 - 2020 Robin Stuart.

Further copyright holders:

- Copyright 2016 - 2020 Harald Oehlmann (CodablockE, Codablock-F)
- Copyright 2004 Adrian Kennard, Andrews & Arnold Ltd, Cliff Hones, Stefan Schmidt (Data Matrix ECC 200)
- Copyright 2004 Grandzebu, KL Chin (XDF417)
- Copyright 2006 Kentaro Fukuchi (QR Code)
- Copyright 2004 Cliff Hones (Reed-Solomon encoder)

Zlib compression library, Copyright 1995-1998 Jean-Loup Gailly and Mark Adler.

The JBIG2 encoder in DynaXDF based on jbig2enc, Copyright 2006 Google Inc. Author: Adam Langley (agl@imperialviolet.org).

DynaXDF contains the RSA Security Inc. MD5 message digest algorithm as well as RC2, RC4, AES 128, and AES 256 encryption algorithms. DynaXDF contains also the Lempel-Ziv-Welch (LZW) compression algorithm, US Patent 4,558,302 Unisys Corporation. The patent expired worldwide in June 2004.

Table of Contents

Data types

DynaXDF is a low level library that uses basic data types only. DynaXDF uses generally no default string class or special C++ extensions such as STL or MFC.

The data types used by DynaXDF are defined as follows (C syntax):

```
typedef unsigned char  BYTE;
typedef signed short   SI16;
typedef unsigned short UI16;
typedef double         FFLOAT; // Obsolete, do not use this data type.

#ifdef _WINDOWS
   #if defined(WIN64) || defined(_WIN64)
      typedef int           SI32;
      typedef unsigned int  UI32;
   #else
      typedef long          SI32;
      typedef unsigned long UI32;
   #endif
#elif (SIZEOF_INT == 4) // declared in drv_conf.h (Linux/UNIX only)
   typedef int           SI32;
   typedef unsigned int  UI32;
#elif (SIZEOF_LONG == 4) // declared in drv_conf.h (Linux/UNIX only)
   typedef long          SI32;
   typedef unsigned long UI32;
#else
   #error "Only 32 bit and 64 bit targets are supported!"
#endif

typedef SI32 LBOOL; // long boolean (0 = false, not 0 = true)
// This data type is provided for C only since this language does not
// support a Boolean data type.
#ifndef __cplusplus
   typedef enum
   {
      false = 0,
      true = 1
   }bool;
#endif
```

The data type char is not explicitly defined but also used by DynaXDF.

SI32 and UI32 must always be a 32 bit integer. It is not possible to use DynaXDF on a target system that does not support a 32 bit integer type, like MS-DOS. DynaXDF can be used on 32 and 64 bit operating systems. The current version was tested with Android, IBM-AIX, HP-UX, iOS, Linux, Mac OS X, Sun-Solaris, Tru64, and MS-Windows.

With very few exceptions, string values returned by DynaXDF are always null-terminated. A string length returned by DynaXDF is always the length excluding the null-terminator.

Var Parameters

```
#ifdef __cplusplus
   #define ADDR &
#else
   #define ADDR *
#endif
```

Functions in DynaXDF, which pass a value to a function parameter are handled differently in C and C++. C does not support the address operator & so that *var* parameters are defined as normal pointers to pointers in C. DynaXDF checks whether a variable or NULL was passed to a function before the function tries to access the variable. However, C++ does not allow to set a parameter to NULL if it was declared with the address operator &.

Structures

Beginning with DynaXDF 2.5 all structures which can be extended in future versions contain the member **StructSize**. This variable must be set to sizeof(StructureName). The structure size is used to identify the version of a structure so that extensions do not break backward compatibility.

The structure size is automatically set in interfaces for C#, Visual Basic, Visual Basic .Net, and Delphi. C/C++ programmers must set this member before the corresponding function can be called:

Example:

```
...
TXDFCMap cmap;
cmap.StructSize = sizeof(TXDFCMap);
XDFGetCMap(XDF, handle, &cmap);
...
```

Multi-byte Strings

Unicode

DynaXDF supports Unicode strings in UTF-16-LE format on little-endian machines and UTF-16-BE on big-endian machines. On target systems which use UTF-32 (LE or BE) as default string format such as Linux or most UNIX OS, all strings must be converted to UTF-16 before passing to DynaXDF.

You can use the predefined macro ToUTF16 to do this.

ToUTF16 is defined as follows:

```
// declared in drv_conf.h (Linux/UNIX, Mac OS X)
#if (SIZEOF_WCHAR_T == 4)
   #define ToUTF16(IXDF, s)(XDFUTF32ToUTF16((IXDF), (UI32*)(s)))
#else                        // UTF-16
   #define ToUTF16(IXDF, s)((s))
#endif
```

This macro calls XDFUTF32ToUTF16() only if the OS uses UTF-32 as Unicode string format.

On operating systems which use already UTF-16, no conversion is applied; the macro will be removed by the compiler. The function XDFUTF32ToUTF16() holds an array of 4 independent string buffers so that the macro can be used in functions which support up to four string parameters. If DynaXDF will ever support a function with more than 4 string parameters, the number of internal string buffers will be incremented.

However, take care when using the macro to initialize string variables of structures which contain more than 6 string members:

Example:

```
SOME_STRUCT myStruct;
myStruct.String1 = ToUTF16(XDF, L"String1"); // OK
myStruct.String2 = ToUTF16(XDF, L"String2"); // OK
myStruct.String3 = ToUTF16(XDF, L"String3"); // OK
myStruct.String4 = ToUTF16(XDF, L"String4"); // OK
myStruct.String5 = ToUTF16(XDF, L"String5"); // OK
myStruct.String6 = ToUTF16(XDF, L"String6"); // OK
myStruct.String7 = ToUTF16(XDF, L"String7"); // Wrong!
```

The seventh call above overrides the string buffer of String1 because only 6 internal string buffers are available. If you need to store more than 6 string variables then you must copy the converted string into another variable!

Unicode File Paths

Unicode file paths are encoded differently depending on the used operating system. While NT based Windows system use UTF-16 encoded Unicode file paths, non-Windows systems use usually UTF-8 encoded Unicode file paths. All DynaXDF functions which open a file convert UTF-16 strings to UTF-8 on non-Windows operating systems. However, to avoid this conversion step it is usually best to use directly the Ansi version of a function and passing an UTF-8 file path to it.

CJK Multi-byte Strings

CJK multi-byte strings contain mixed 8 bit / 16 bit character codes. A CJK string can be defined as an Ansi string (data type char*) and as multi-byte string (data type UI16*). The multi-byte format

uses two bytes for every character and the byte ordering of the CPU must be considered to get correct results on little-endian and big-endian machines.

However, the multi-byte format is only supported in combination with native CJK fonts and character sets (cpBig5, cpShiftJIS, cpGB2312 and so on), see SetFont() for further information.

The Ansi format is the usual format for CJK strings and supported by all CJK code pages.

When using CJK to Unicode code pages, DynaXDF must convert the incoming CJK string to Unicode before it can be used with the selected font. The required conversion algorithms are only available in the Ansi version of a string function. Because of this it is not possible to use the multi-byte format with CJK to Unicode code pages.

Data types used by different programming languages

Not all programming languages support all data types which are available in C or C++. The following table describes the data types which are used by a specific programming language. Types in black color are not natively supported.

C type	C++ type	Delphi type	VB type	VB .Net type	C#
char*	char*	PAnsiChar	String	String	String
UI16*	UI16*	PWideChar	String	String	String
UI32*	UI32*	Pointer	N/A	N/A	N/A
SI32	SI32	Integer	Long	Integer	int
UI32	UI32	Cardinal	Long	Integer	int
double	double	Double	Double	Double	double
LBOOL	LBOOL	LongBool	Long	Integer	lbool (Int32)
bool	bool	Boolean	Boolean	Boolean	bool
void*	void*	Pointer	Long	IntPtr	IntPtr

VB .Net and C# support also unsigned data types such as unsigned integer and so on. However, the .Net framework requires always an explicit conversion if an unsigned type should be passed to a signed type or vice versa. Because of this, the unsigned data types are used for very few functions in VB .Net and C#.

Prior versions of DynaXDF used the abbreviations LONG, ULONG, SHORT, and USHORT in the C/C++ interfaces. For compatibility reasons these data types are still defined on 32 bit Windows, Linux, and UNIX. However, these declarations conflict with existing declarations on 64 bit Windows operating systems so that these types should no longer be used.

Exception handling

The DynaXDF library uses no native exception handling. It is not required to leave a function block after an error occurred. All DynaXDF functions are leaved immediately when a fatal error occurred without displaying further error messages.

The library always holds its internal state consistent, regardless what happens.

Exception handling in C, C++, C#, Delphi

Error messages and warnings are passed to a callback function (see SetOnErrorProc()) if set. If no callback function is set, you must check the return value of important functions. Negative return values indicate by default that an error occurred. Call GetErrorMessage() to get the last error message in this case.

The Delphi interface uses native language exceptions in the following reasons:

- When loading the dynaXDF.dll (Cannot find dynaXDF.dll)
- When loading a DynaXDF function (Cannot find function: …)
- When creating a new instance of the wrapper class TXDF (Out of memory).

Only these three exceptions can occur when using DynaXDF with Delphi. All other errors do not raise an exception; the error callback function is called instead if any.

The error callback function is defined as follows (C/C++):

```
typedef SI32 XDF_CALL TErrorProc(
   const void* Data,       // User defined pointer
   SI32 ErrCode,           // Error code
   const char* ErrMessage, // Error message
   SI32 ErrType)           // Error type

#define XDF_CALL __stdcall // Windows only, otherwise empty
```

All callback functions contain a parameter named "Data" which holds a user defined pointer. *Data* can be set with SetOnErrorProc(). If you don't need this pointer, set it to NULL. The pointer *Data* is always passed unchanged to the callback function.

ErrCode is a positive error number starting at zero. *ErrType* is a bit mask to determine what kind of error occurred. The following constants are defined:

```
#define E_WARNING      0x02000000
#define E_SYNTAX_ERROR 0x04000000
#define E_VALUE_ERROR  0x08000000
#define E_FONT_ERROR   0x10000000
#define E_FATAL_ERROR  0x20000000
#define E_FILE_ERROR   0x40000000
```

At time of publication only one flag is set at any one time. Future versions maybe set multiple flags, e.g. E_SYNTAX_ERROR and E_WARNING.

Because of this, it is required to mask out the error type:

```
if (ErrType & E_SYNTAX_ERROR)
{
   // some code
   return 0; // continue processing
}
```

If the error callback function returns a value other than zero, processing stops immediately.

All callback functions must use the correct calling convention. The C definition above contains the macro `XDF_CALL` that is defined as __stdcall under Windows. Note that a wrongly defined calling convention causes an access violation!

When using DynaXDF with C or C++, you can change the calling convention to __fastcall or __cdecl by changing the macro XDF_CALL in the main interface dynaXDF.h and in the project settings. However, standard call is strongly recommended. Note that the library must be recompiled when changing the calling convention. Delphi, Visual Basic, and VB .Net require standard call!

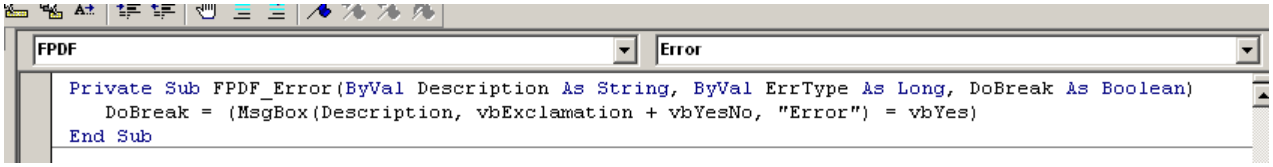
Exception handling in Visual Basic, Visual Basic .Net

The Visual Basic wrapper class CXDF uses events instead of callback functions to pass error messages and warnings to the user; native language exceptions are not used.

The usage of the event functions is quite easy; just declare a local instance variable of the wrapper class CXDF in the Option Explicit section of the unit as follows:

```
Private WithEvents FXDF As CXDF
```

The command above enables the event support of the class CXDF. The instance variable FXDF is now listed in the left combo box of the VB code editor.



The right combo box contains the available events, when selecting the event "Error" VB adds automatically an empty event procedure to your source code:

```
Private Sub FXDF_Error(ByVal Description As String, ByVal ErrType As
Long, DoBreak As Boolean)
   ' Add your code here
End Sub
```

Now you can enter some code that should be executed when the event is fired. Note that you must still create an instance of the class CXDF before a DynaXDF function can be executed (see Language bindings/Visual Basic for further information).

The handling of events in VB .Net is exactly the same as in VB 6.0.

Special issues in Visual Basic and .Net

The usage of events in Visual Basic or VB .Net is quite easy; however, there is a special behaviour that must be taken into account when developing VB applications. When using the DoEvents procedure in a VB function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is running (e.g. break processing). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaXDF functions while a previous call is still running. This is normally no problem but it is impossible to execute an event function inside of a cloned function. When DynaXDF tries to raise an event inside the cloned function an access violation occurs and VB crashes. VB .Net does not crash but raises a System.NullReference exception in that case.

To avoid such problems check whether the function is still running:

```
Option Explicit
Private WithEvents FXDF As CXDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click()
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaXDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above simply checks whether a previous call of the function is running before the function can be executed again.

Customized Exception handling

By default, only fatal errors will stop processing. Warnings, syntax errors and so on are all ignored. You can customize the exception handling to your own requirements with the property Get/SetOnErrorMode(). With the following constants you can determine what kind of error should be treated as fatal error:

```
typedef SI32 TErrMode;
#define emIgnoreAll    0x00000000 // default
#define emSyntaxError  0x00000001
#define emValueError   0x00000002
#define emWarning      0x00000004
#define emFileError    0x00000008
#define emFontError    0x00000010
#define emAllErrors    0x0000FFFF
#define emNoFuncNames  0x10000000 // Do not output function names
```

If a specific flag is set, DynaXDF treats this error type as fatal error; the internal resources will be freed and all changes made before are lost.

DynaXDF never produces a damaged XDF file if a warning or error message was ignored, but certain functions may be not executed. For example, if SetFont() cannot find the selected font, it returns with a warning and no font is set, the active font (if any) is left unchanged.

If no other font was set before, it is not possible to output text. All text functions also return then with a warning because there is no active font, but nothing more happens than a warning is issued.

When a fatal error occurred, all functions are leaved immediately. No further warnings or error messages are displayed. It is not possible to execute a function (except global properties which do not change XDF objects directly) after a fatal error occurred.

There is no need to check the return value of each function, and there is no need to leave a function block after a fatal error occurred. The internal error flag is cleared when CreateNewXDF() is called the next time.

The special flag ***emNoFuncNames*** names can be used to avoid the output of the function name in error messages. Error messages start normally always with the function name in which the error occurred. While this information is useful during development, it is often not useful in an end user application.

Remarks:

The constants are defined as enum in Visual Basic, VB .Net, and C#.

Custom Library Changes

This section is only of interest if you have a copy of the source codes. If you use a version without source codes you can skip this chapter.

Compiler Switches

DynaXDF supports several compiler switches to disable unnecessary features. The following macros disable or enable one of the image libraries used by DynaXDF as well as other features. The macros are defined in the header file /main/drv_type.h.

```
#define DRV_SUPPORT_AES  1 // AES encryption and decryption
#define DRV_SUPPORT_BCDE 1 // about 400 KB -> Barcode engine
#define DRV_SUPPORT_CJK  1 // about 150 KB (CJK to Unicode conversion)
#define DRV_SUPPORT_EMF  1 // EMF Converter
#define DRV_SUPPORT_GIF  1 // about 1 KB
#define DRV_SUPPORT_IMP  1 // about 50 KB (XDF import)
#define DRV_SUPPORT_JP2K 1 // about 180 KB
#define DRV_SUPPORT_JPEG 1 // about 90 KB (no effect if TIFF is enabled)
#define DRV_SUPPORT_PGM  1 // PBM, PGM, PNM, PPM Image formats, ~1 KB
#define DRV_SUPPORT_PNG  1 // about 100 KB
#define DRV_SUPPORT_PSD  1 // about 1 KB
#define DRV_SUPPORT_RAS  1 // about 700 KB -> Rendering engine
#define DRV_SUPPORT_RC4  1 // RC4 encryption and decryption
#define DRV_SUPPORT_SIGN 1 // Self sign signatures -> AiCrypto Library
#define DRV_SUPPORT_TIFF 1 // about 310 KB
```

To disable a specific feature set the constant to zero or comment it out. Note that the TIFF library uses also the JPEG library. Because of this, disabling the JPEG library only does not reduce the library size.

The following constants are used by WriteFText() (defined in dynaXDF.h).

```
#define XDF_MAX_LIST_COUNT 6    // Maximum count of nested list levels
#define XDF_LIST_SEP_WIDTH 10.0 // Default list separator with
#define DEFAULT_LIST_CHAR  159  // Default list character
#define XDF_LIST_FONT "Wingdings-Regular" // Default list font
```

The list font must be the PostScript name of the font (see SetFont() for further information). When changing the list font you may also change the default list character. When using DynaXDF under Linux or UNIX you may define a font that is available in one of your font search directories.

Note that the list font can be overridden at runtime with the function SetListFont(). So, it is usually better to load the font at runtime with SetListFont() since you can properly handle cases in which the font cannot be found.

Main object types

In XDF, two basic object types can be created, ***resource objects*** such as fonts, images and so on which are used by content streams and ***global objects*** such as annotations, bookmarks, form fields and so on. Global objects can use resource objects but not vice versa.

In most cases both object types are defined as normal classes, which contain their own constructors and destructors to initialize and destroy allocated memory.

Global objects can be deleted or marked as deleted at runtime. Resource objects must never be deleted if the object was already used.

General design requirements

We describe here only the general rules which must be taken into account when extending DynaXDF with certain features. We do not explain how the entire library works; this would fill an entire book. To understand how an object must be written to the file we recommended that you debug especially the function CloseFile(). All objects must be prepared for writing in a two stage phase to reserve object numbers. The first stage assignes object numbers to all objects and the second run writes all objects to file. Objects must be written in the exact order in which they were previously prepared for writing.

A XDF file is described in memory as large set of classes which can be referenced or used multiple times by other classes. Due to the references which are stored in certain classes we must define a strict set of rules so that no exception occurs if an object must be deleted:

1. The owner of all resources and global objects is CXDF. No other class is permitted to destroy an object class or change its values.
2. All object classes must be derived from CBaseObject. This class holds the object number as well as several flags to determine whether the object was used, created, or already written, and whether it is part of the first page. This class contains the function CreateObject() which must be called in CXDF::PreparePageObjects() if the object is part of a page. If the object is not included in a page object then CreateObject() must be called in CXDF::PrepareObjects().
3. All classes which hold pointers to other object classes must check the "Used" flag before writing the object data to the file (GetUsed() is a member of CBaseObject and returns true if the used flag was set).
4. All classes must be well initialized so that the class can be deleted at any time without causing memory leaks or other unwanted side effects.
5. No object class is permitted to unset the "Used" flag of other object classes.
6. Page resources such as fonts, images, templates and so on must **NEVER** be deleted at runtime and their "Used" flag must **NEVER** be unset.
7. All resource classes must be derived from CBaseResource.
8. Object classes must set the "Used" flag of the resource object, if the class is used by this object.

9. The "Used" flag of non-page resources can be unset at runtime with SetUnUsed(). This will mark the class as an unused or deleted object. When writing the file, such an object must be ignored.
10. Never delete an object if you don't know exactly what you are doing. If a destroyed object is referenced in any other object, the library will crash.
11. If a **used** page resource is deleted or if the "Used" flag is unset at runtime when it was already set, the resulting XDF file will be damaged.
12. Make sure that the function FreeXDF() can be called at any time without causing memory leaks or other unwanted side effects.

Requirements to add your own code to DynaXDF

If you want to make your own features permanent, you are welcome to send us your source codes including a description what kind of feature it enables.

However, we cannot accept any old code; your code must be properly written in C++ and tested with certain operating systems and compilers. It must not produce warnings of any kind and it must not use external libraries, except those are already used by DynaXDF (see drv_type.h for a full list).

Basic C functions such as strcpy, strcat, memcpy and so on, as well as templates from the STL are **NOT** allowed to use. Take a look into drv_base_func.h, drv_templ.h, or XDFutils.h before using an external function or template. The implementations used by DynaXDF are faster and work on all operating systems in the same way.

Windows GDI functions are normally not permitted too. However, under certain circumstances GDI functions are permitted. DynaXDF uses already a few GDI functions to convert WMF files to EMF or to raster EMF files. Windows specific code must be encapsulated into a #ifdef _WINDOWS section.

If your code handles strings, it must **NOT** use an external string class which is available in any standard C++ library. Use the DynaXDF class CString or CXDFString() instead. These classes support many XDF specific functions.

Your code should be tested with Microsoft Visual C++ 6.0 **AND** Visual Studio 2005 or higher. If possible, you should test your code also with GCC 4.0 or higher under Linux or Mac OS X.

The following important defines are available depending on the operating system and characteristics of the target CPU:

Constant	Test code	Comment
_WINDOWS	#ifdef _WINDOWS	Set on Windows.
MAC_OS_X	#ifdef MAC_OS_X	Set on Mac OS X.
DRV_BIG_ENDIAN	#if (DRV_BIG_ENDIAN == 1)	Endian configuration.
VS_2005_OR_HIGHER	#if (VS_2005_OR_HIGHER == 1)	Functions which are considered as unsafe in Visual Studio 2005 or higher must be replaced with their safe versions. Use this macro to test the compiler version.

To get the full list of available defines take a look into the header file /main/drv_type.h. On non Windows operating systems the configure script creates also the header file /main/drv_conf.h which contains many OS specific defines. Note that this header file can only be included on non Windows operating systems.

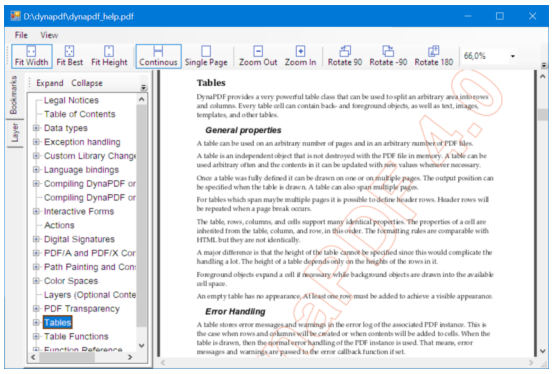
Note also that the class structure in DynaXDF is not fixed. Changes can be made without further notice at any time. So, you cannot assume that a class or member of a class that exists today exists in a newer version too. Although most classes and templates do seldom change, changes can always occur!

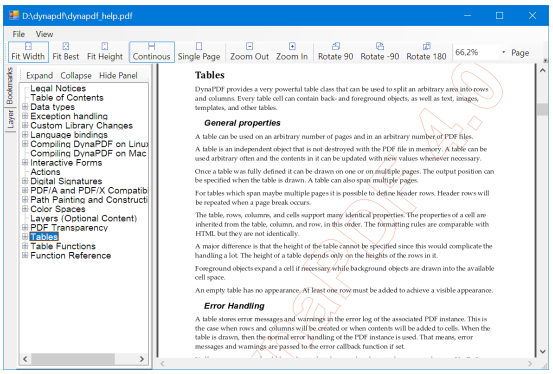
To avoid dependencies to a specific source code version it is usually best to ask us for the best strategy if you want to add certain features to the library.

DPI Aware .Net applications

The difference of dpi aware applications in comparison to regular applications is that the latter one do not properly scale on high resolution displays. Blurry text is an indication that an application is not dpi aware.

C# XDF viewer example with disabled and enabled high dpi support:





As you can see above the treeview control does not properly scale. The same control looks much better when rendered in a lower resolution. Such issues require some fine tuning for optimal results.

A user can of course right click on the executable and change the high dpi settings in the compatibility section of the application properties. This works pretty well with Windows 10 Version 2004 but every major version of Windows 10 behaves differently and we have currently 11 major versions of Windows 10!

A bit annoying is also the fact that these settings do generally not work if the application was copied to a RAM disk on Windows 10 Version 2004. This was no issue in earlier versions.

So, it would be much better to enable high dpi support without additional user interaction. In earlier days high dpi support needed to be enabled with a dpi aware manifest. This worked for a while until the way was changed how high dpi support must be enabled.

The new way of enabling high dpi support is to add a section to the app.config file.

Enabling High DPI support in Windows Forms Apps

To get high dpi support enabled in Windows Forms Apps you must first change the version of the used .Net framework to 4.7 or higher.

If you do this in the settings of the XDF viewer example, for example, then you'll notice that the required app.config fille will be added automatically to the solution. Double click on it and copy the code marked in yellow into the configuration tag:

```
<?xml version="1.0" encoding="utf-8"?><configuration>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/></startup>
<System.Windows.Forms.ApplicationConfigurationSection>
   <add key="DpiAwareness" value="PerMonitorV2" />
</System.Windows.Forms.ApplicationConfigurationSection>
</configuration>
```

The above settings are for Windows Forms Apps. Note that this is how it works today, tomorrow you must maybe change other settings.

WPF and UWP apps come out of the box with high dpi support but using native code for UI elements in WPF or UWP is difficult and not recommended. WPF and UWP apps function more or less like a web site. When speed is no concern it is better to use pure .Net code in these environments.

Scrolling issues on high dpi devices

Scrolling issues are one of the major issues on high dpi devices. Almost every application scrolls differently on Windows 10. Some scroll way to fast, others scroll too slow, and again others are stuttering or have weird scroll behavior.

To avoid scrolling issues is probably best to get users a chance to adjust the scrolling speed in your application since the settings of the system are often not helpful.

If you use the rendering control of DynaXDF or the page cache then you can adjust the scroll line delta with SetScrollLineDelta(). Smooth scrolling is not yet supported.

Static contents in dpi aware applications

Something that must be considered when turning on high dpi support is that static contents like the application icon or cursors must be delivered in different sizes since no scaling occurs.

Especially cursors are critical because an unscaled cursor made for Windows 7, for example, becomes very small. Microsoft recommends to add at least icons and cursors for the resolutions 96, 120, 144, and 196 DPI for desktop apps. You'll find many tools in the internet which help to create icons and curors in different sizes.

Language bindings

Differences between DynaXDF interfaces

DynaXDF can be used with most programming languages which support standard DLLs. The usage of DynaXDF is nearly identical in all programming languages. However, this help file describes all DynaXDF functions in C syntax.

All DynaXDF functions contain an instance pointer of the active XDF instance `(const PXDF* IXDF)` as first parameter. This pointer is hidden for the user in the programming languages Visual Basic, Visual Basic .Net, Visual C#, and Delphi. We deliver native wrapper classes for these programming languages which handle the XDF instances automatically.

However, this help file describes the raw API of DynaXDF that is used by the programming languages C and C++. The C/C++ interface is a direct interface that does not encapsulate the DynaXDF API into a class or other structures to improve processing speed.

The usage of DynaXDF is almost identical in all programming languages; you must only consider that the instance pointer IXDF is contained in the C/C++ interface only.

Visual Basic, Visual C#, or Delphi users must create an instance of the wrapper class CXDF or TXDF in Delphi before a DynaXDF function can be executed. The instance of the wrapper class must also be deleted by calling the destructor of the class.

C or C++ programmers must create a XDF instance with the function XDFNewXDF() before a DynaXDF function can be executed. This instance must be deleted with the function XDFDeleteXDF() when it is no longer needed.

We tried also to consider the specific requirements for each programming language so that DynaXDF can always be used without limitations. This causes slightly differences because of the differences between programming languages. For instance, the Visual Basic interfaces uses events instead of callback functions because the usage of callback functions is more complicated in VB as in other programming languages.

Embarcadero C++ Builder

To use DynaXDF with Embarcadero's C++ Builder, proceed as follows:

1. Open a new project or your favourite project in C++ Builder.
2. Include the header file ***/include/C_CPP/dynaXDF.h*** into the units which will use DynaXDF functions.
3. Add the import library ***/borland_lib/dynaXDF.lib*** to your project.
4. Copy the ***dynaXDF.dll*** into a Windows search path (e.g. Windows/System32) or into the output directory.
5. Finished!

If you want to use DynaXDF with a C++ project, add the line "using namespace DynaXDF;" after including the header file dynaXDF.h.

The usage in C is essentially the same as in C++, with the exception that the namespace DynaXDF must not be declared.

A XDF instance can be used to create an arbitrary count of XDF files. All used resources are automatically freed when the XDF file is closed (except when the XDF file was created in memory). To improve processing speed, use one instance as long as possible.

If you want to use DynaXDF with an older version of C++ Builder, you may rebuild the lib file by using implib.exe. Implib is delivered with C++ Builder; you find it in the *Bin* directory of your C++ Builder installation directory.

To create a new import library run implib from the command line as follows:

```
implib dynaXDF.lib dynaXDF.dll
```

DynaXDF uses standard call as calling convention. C++ Builder requires for that calling convention no underscores before function names. Because of this the option -a must not be used to build the import library (see your C++ Builder help for further information).

Copy *implib* and the dynaXDF.dll into the same directory and execute the command above.

Example (C++ Builder):

```
// Standard includes by all C++ Builder projects
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

// Include the DynaXDF header file (change the path if necessary)
```

```
#include "dynaXDF.h"
// All data types and functions are declared in the namespace DynaXDF.
using namespace DynaXDF;
// First, we declare an error callback function that is called by
// DynaXDF if an error occurred.
SI32 XDF_CALL XDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   char errMsg[XDF_MAX_ERR_LEN + 30];
   // An error message returned by DynaXDF is a pointer to a null-
   // terminated static string.
   sprintf(errMsg, "%s\n\nAbort processing?\n", ErrMessage);
   if (MessageDlg(errMsg, mtError,
   TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes)
      return -1; // break processing
   else
      return 0;  // ignore the error
}

// Place a button on the form and double click on it. Add the
// following code to the function.
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   void* XDF = XDFNewXDF(); // Create a new XDF instance
   if (!XDF) return; // Out of memory?
   // Set the error callback first
   XDFSetOnErrorProc(XDF, NULL, XDFError);
   XDFCreateNewXDF(XDF, "c:/myfirst.XDF");
   XDFSetDocInfo(XDF, diSubject, "My first C++ output");
   XDFSetDocInfo(XDF, diProducer, "C++ Builder test application");
   XDFSetDocInfo(XDF, diTitle, "My first C++ output");
   // We want to use top-down coordinates
   XDFSetPageCoords(XDF, pcTopDown);

   XDFAppend(XDF);
   XDFSetFont(XDF, "Arial", DynaXDF::fsItalic, 40.0, true, cp1252);
   XDFWriteFText(XDF, DynaXDF::taCenter, "My first C++ output!");
   XDFEndPage(XDF);

   XDFCloseFile(XDF); // Close the file and free all used resources
   XDFDeleteXDF(XDF); // Do not forget to delete the XDF instance
}
```

Microsoft Visual C++

To use DynaXDF with Microsoft's Visual C++, proceed as follows:

1. Open a new project, or your favourite project, in Visual Studio.
2. Include the header file ***/include/C_CPP/dynaXDF.h*** into the units which will use DynaXDF functions.
3. Add the import library ***/mvs_lib/dynaXDF.lib*** or ***/mvs_lib/dynaXDFm.lib*** to your project.
4. Copy the ***dynaXDF.dll*** or ***dynaXDFm.dll*** into a Windows search path (e.g. Windows/System32).
5. Finished!

DynaXDF was compiled and developed with Microsoft Visual Studio 2017 but the library can be used with older version too. Two versions of the library are delivered:

- `dynaXDF.dll  // Compiled with Multithreaded`
- `dynaXDFm.dll // Compiled with Multithreaded DLL`

The **dynaXDFm.dll** depends on the **Visual Studio Redistributable** package that can be downloaded from our website. The latest version can also be downloaded from the Microsoft website. Visual Studio 2015, 2017, 2019, and 2022 use all the same redistributable package.

If possible, use the dynaXDF.dll since this version has no additional dependency.

If you want to use DynaXDF with a C++ project, add the line "using namespace DynaXDF;" after including the header file dynaXDF.h.

The usage in C is essentially the same as in C++, with the exception that the namespace DynaXDF must not be declared.

A XDF instance can be used to create an arbitrary number of XDF files. All used resources are automatically freed when the XDF file is closed (except when the XDF file was created in memory, in this case FreeXDF() must be called when finish). To improve processing speed, use one instance as long as possible.

Most examples are written in C or C++.

Example (C++):

```
// Include the DynaXDF header file (change the path if necessary)
#include "dynaXDF.h"
// All data types and functions are declared in the namespace DynaXDF.
using namespace DynaXDF;
// First, we declare an error callback function that is called by
// DynaXDF if an error occurred.
SI32 XDF_CALL XDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
```

```
{
   printf("%s\n", ErrMessage);
   return 0; // Any other return value break processing
}
// Place a button on the form and double click on it. Add the
// following code to the function.
int main(int argc, char* argv[])
{
   void* XDF = XDFNewXDF(); // Create a new XDF instance
   if (!XDF) return 2; // Out of memory?
   // Set the error callback first
   XDFSetOnErrorProc(XDF, NULL, XDFError);
   XDFCreateNewXDF(XDF, "c:/myfirst.XDF");
   XDFSetPageCoords(XDF, pcTopDown); // We use top-down coordinates
   XDFAppend(XDF);
   XDFSetFont(XDF, "Arial", fsItalic, 40.0, true, cp1252);
   XDFWriteFText(XDF, taCenter, "My first C++ output!");
   XDFEndPage(XDF);
   XDFCloseFile(XDF); // Close the file and free all used resources
   XDFDeleteXDF(XDF); // Do not forget to delete the XDF instance
}
```

Microsoft Visual Basic 6.0

The usage of DynaXDF with Visual Basic is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CXDF to make the usage easier. The instance pointer IXDF which is used by every DynaXDF function is hidden for the user in Visual Basic. The instance pointer is controlled by the wrapper class so that you don't need to create XDF instances manually.

To use DynaXDF with Visual Basic proceed as follows:

- Add the file ***/include/Visual_Basic/DynapXDFInt.bas*** to your project (menu Project/Add Module/Existing…).
- Add the file ***/include/Visual_Basic/DynapXDFInt2.bas*** to your project (menu Project/Add Module/Existing…).
- Add the file ***/include/Visual_Basic/CXDF.cls*** to your project (menu Project/Add Class Module/Existing…).
- Add the file ***/include/Visual_Basic/IXDFCallback.cls*** to your project (menu Project/Add Class Module/Existing…).
- If you want to use the table class then add also the file ***/include/Visual_Basic/CXDFTable.cls*** to your project (menu Project/Add Class Module/Existing…).
- Finally, make sure that the ***dynaXDF.dll*** can be found by Visual Basic in debug mode; just copy the DLL into Windows/System32 or into Windows/SysWow64 on a 64 bit system, finished!

Note that Visual Basic supports the 32 bit dynaXDF.dll only. If you work on a 64 bit OS then copy the library into Windows/SysWow64. Yes, this is the right folder for 32 bit DLLs!

All DynaXDF functions are encapsulated in the wrapper class CXDF. This class makes sure that the library can be used without limitations and programming is more comfortable since you can work with a native VB class. You don't need to consider specific return values of the DLL, the class converts special data types automatically to VB data types.

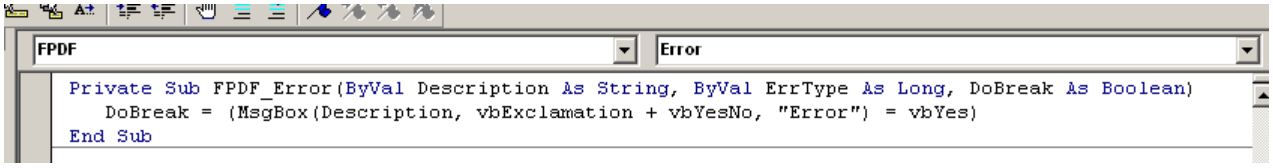
Exception handling in Visual Basic

The standard exception handling of DynaXDF uses a callback function to pass error messages and warnings to the client application. However, in Visual Basic we use events instead. The usage of events is quite easy and frees you from dealing with DynaXDF API function calls.

To enable the event support of the wrapper class CXDF declare a local instance variable as follows:

```
Option Explicit
Private WithEvents FXDF As CXDF 'Enable event support
```

The instance variable FXDF is now listed in the left combo box of the VB code editor.



The right combo box contains the available events, when selecting the event "Error" VB adds automatically an empty event procedure to your source code:

```
Private Sub FXDF_Error(ByVal Description As String, ByVal ErrType As
Long, DoBreak As Boolean)
   ' Add your code here
   ' To break processing after an error occurred set the variable
   ' DoBreak to True
End Sub
```

Now you can enter some code that should be executed when the event is fired. Note that you must still create an instance of the class CXDF before a DynaXDF function can be executed (see the example below).

Note also that VB exceptions are not used so that an error block will normally never be executed. The following declaration has effect in the means of the DynaXDF exception handling:

```
On Error GoTo errXDF
...
' Call some DynaXDF functions...
: errXDF
If Err.Number <> 0 Then
   MsgBox Err.Description
   Exit Sub
End If
```

The error block cannot be executed because DynaXDF does never raise an exception (except the class constructor). If an error occurred, the event "Error" is raised instead. However, there are still cases in which a VB exception can occur, e.g. when passing an empty array to a function that requires some values in it. So, the code should still be encapsulated in an error block. All you need to know that this error block does not affect the normal exception handling of DynaXDF.

The DoEvents problem

The usage of events in Visual Basic is quite easy but there is a special behaviour that must be taken into account when developing VB applications. When using the DoEvents procedure in a VB function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is still running (e.g. press a break button). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaXDF functions while a previous call is still running. This is normally no problem but it is impossible to execute an event functions inside of a cloned function. When DynaXDF tries to raise an event inside the cloned function an access violation occurs and VB crashes.

To avoid such problems check whether the function is still running:

```
Option Explicit
Private WithEvents FXDF As CXDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click()
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaXDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above checks whether a previous call of the function is running before the function can be executed again; a quite simple but effective solution that makes your application stabile.

Example:

```
Option Explicit
Private WithEvents FXDF As CXDF ' Enable event support
Private Sub Form_Load()
   ' We hold one instance of the class CXDF in memory
   Set FXDF = new CXDF
   If FXDF Is Nothing Then
      MsgBox "Out of memory!", vbCritical, "Fatal error"
   End If
End Sub
Private Sub Form_Terminate()
   ' Delete the class instance
   Set FXDF = Nothing
End Sub
Private Sub Command1_Click()
   FXDF.CreateNewXDFA "c:/vbout.XDF"
   FXDF.SetDocInfoA diAuthor, "Jens Boschulte"
   FXDF.SetDocInfoA diSubject, "My first VB output"
   FXDF.SetDocInfoA diTitle, "My first VB output"
   FXDF.Append
   FXDF.SetFont "Arial", fsItalic, 30#, True, cp1252
```

```
   FXDF.WriteFTextA taCenter, "My first VB output"
   FXDF.EndPage
   FXDF.CloseFile
End Sub
' Error event procedure
Private Sub FXDF_Error(ByVal Description As String, ByVal ErrType As
Long, DoBreak As Boolean)
   DoBreak = (MsgBox(Description, vbExclamation Or vbYesNo,
             "Error") = vbYes)
End Sub
```

Visual Basic .Net

The usage of DynaXDF with Visual Basic .Net is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CXDF to make the usage easier. The instance pointer IXDF that is used by every DynaXDF function is hidden for the user in VB .Net. The instance pointer is controlled by the wrapper class so that you don't need to create XDF instances manually.

To use DynaXDF with VB .Net proceed as follows:

- Add the file ***/include/Visual_Basic_Net/CXDF.vb*** to your project (menu Project/Add Existing Element…).
- Finally, make sure that the ***dynaXDF.dll*** can be found; just copy the DLL into Windows/System32, finished!

64 Bit Applications

With VB .Net you can develop 32 bit and 64 bit applications. One thing that must be considered is that the target CPU type in Visual Studio must **not** be set to *UseAny*. This is impossible since you can either link the 32 bit dynaXDF.dll or the 64 bit version but not both.

So, a 32 bit and 64 bit version must be compiled separately. Another thing that is often misunderstood is the right system directory for the dynaXDF.dll. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynaXDF.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Visual Studio is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Note:

Visual Studio .Net copies the interface files into your project directory if the option “Link file” is not selected when adding the files to your project. Make sure that you always link the file to your project. Otherwise you must update the interface files manually whenever you install a newer version of DynaXDF.

All DynaXDF functions are encapsulated in the wrapper class CXDF. This class makes sure that the DynaXDF functions can be used without limitations and programming with DynaXDF becomes more comfortable. You don't need to consider specific return values of the DLL; the class converts API data types automatically to VB .Net data types.

VB .Net supports more data types than VB 6.0 but the usage of the new integer data types is more complicated since signed integer values cannot be passed to unsigned integer values without explicit conversion.

To make the usage of DynaXDF less complicated the VB .Net interface uses only basic data types which are already available since VB 6.0. An exception is the definition of pointers. VB .Net supports the new Data IntPtr that is a variable pointer type with a length of 32 bit on a 32 bit Windows machine and 64 bit on a 64 bit Windows machine. This data type is used for all pointers so that the .Net interface can also be used on a 64 bit Windows machine. Note that the 64 bit version of DynaXDF must be used in this case.

Data types used by DynaXDF

DynaXDF uses a large set of enums and other data types which are mostly declared within the class CXDF. However, a few data types are declared in the file DynaXDFInt.vb, this must be taken into account when developing VB .Net applications. If you cannot find a data type in the class CXDF then take a look at the file DynaXDFInt.vb or DynaXDFInt2.vb.

Exception handling in VB .Net

The exception handling in Visual Basic .Net is the same as in Visual Basic 6.0. The class CXDF uses per default events instead of callback functions. This makes the usage easier and frees you from handling with unmanaged data types. Please take a look into Visual Basic Exception handling for a detailed description.

The DoEvents problem

The usage of events in VB .Net is quite easy; however, there is a special behaviour that must be taken into account when developing .Net applications. When using the DoEvents procedure in a function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is still running (e.g. press a break button). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaXDF functions while a previous call is still running. This is normally no problem but when using events the event functions become invalid. This is the same behaviour as in Visual Basic 6.0 with the exception that .Net does not crash, a System.NullReferenceException is raised instead.

It is not clear why this exception occurs, it seems that this is a general bug in the event handling of Visual Basic 6.0 and VB .Net. A native programming language like C/C++ or Delphi would never cause an access violation or exception here.

However, to avoid such problems check whether the function is still running:

```
Private WithEvents FXDF As CXDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles Command1.Click
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaXDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above checks whether a previous call of the function is running before the function can be executed again. A quite simple but effective solution that makes your application stabile.

Example (Visual Basic .Net):

```
Private WithEvents FXDF As CXDF 'Enable event support

Private Sub Form1_Load(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles MyBase.Load
   ' We hold one instance of the class CXDF in memory
   FXDF = New CXDF()
End Sub

Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles Command1.Click
   FXDF.CreateNewXDFA("c:/vbout.XDF")
   FXDF.SetDocInfoA(CXDF.TDocumentInfo.diAuthor, "Jens Boschulte")
   FXDF.SetDocInfoA(CXDF.TDocumentInfo.diSubject, "My first VB output")
   FXDF.SetDocInfoA(CXDF.TDocumentInfo.diTitle, "My first VB output")

   FXDF.Append()
   ' The data type TFStyle is defined in DynaXDFInt.vb
   FXDF.SetFontA("Arial", TFStyle.fsItalic, 30.0, True, CXDF.TCodepage.cp1252)
   FXDF.WriteFTextA(CXDF.TTextAlign.taCenter, "My first VB .Net output")
   FXDF.EndPage()
   FXDF.CloseFile()
End Sub

' Error event procedure
Private Sub FXDF_XDFError(ByVal Description As String, ByVal ErrType
As Integer, ByRef DoBreak As Boolean) Handles FXDF.XDFError
   DoBreak = (MsgBox(Description, _
```

```
             MsgBoxStyle.Exclamation Or _
             MsgBoxStyle.YesNo, _
            "Error") = MsgBoxResult.Yes)
End Sub
```

Visual C#

The usage of DynaXDF with Visual C# is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CXDF to make the usage easier. The instance pointer IXDF that is used by every DynaXDF function is hidden for the user in C#. The instance pointer is controlled by the wrapper class so that you must not create XDF instances manually.

.Net Core compatibility

.Net Core applications can be compiled with Visual Studio for Windows, Linux, and macOS. In order to use DynaXDF on these operating systems, copy the DynaXDF DLL or shared library into the deploy folder. The library can be found in the dynaXDF directory on Windows or download the corresponding Linux or macOS archive respectively:

- `Windows 32 bit:  /dynaXDF/win32/dynaXDF.dll`
- `Windows 64 bit:  /dynaXDF/win64/dynaXDF.dll`
- `Linux 32/64 bit:  /dynaXDF/dynaXDF/libdynaXDF.so`
- `macOS 64 bit:  /dynaXDF/dynaXDF/libdynaXDF.dylib`

AOT deployment is not supported.

Using DynaXDF with Visual Studio

To use DynaXDF with Visual C# proceed as follows:

- Add the file ***/include/Visual_C#/CXDF.cs*** to your project (menu Project/Add Existing Element…).
- Make sure that the ***dynaXDF.dll*** can be found. On the development machine execute the msi installer (dynaXDF.msi). The installer copies the the 32 and 64 bit library into Windows/System32 and Windows/SysWow64. On Windows 7 or earlier it is also possible to copy the library manually into the system folders but this is not possible on Windows 10.
- Note that an installation into the system folders is recommended on a development machine only. On the target system the DLL should be copied into the application folder to avoid issues with different DynaXDF versions which might be installed on the system.

64 Bit Applications

With C# you can develop 32 bit and 64 bit applications. One thing that must be considered is that the target CPU type in Visual Studio must **not** be set to *UseAny*. This is impossible since you can either link the 32 bit dynaXDF.dll or the 64 bit version but not both.

So, a 32 bit and 64 bit version must be compiled separately. Another thing that is often misunderstood is the right system directory for the dynaXDF.dll. If you develop a 32 bit

application on a 64 bit Windows version then copy the 32 bit version of the dynaXDF.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Visual Studio is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Note:

Visual Studio .Net copies the interface file CXDF.cs into your project directory if the option "Link file" is not selected when adding the file to your project. Make sure that you always link the files to your project. Otherwise you must update the interface manually whenever you install a newer version of DynaXDF.

All DynaXDF functions are encapsulated in the wrapper class CXDF. This class makes sure that the DynaXDF functions can be used without limitations and programming with DynaXDF becomes more comfortable. You don't need to consider specific return values of the DLL; the class converts API data types automatically to C# data types.

However, C# uses a very restrictive data type handling that causes that already signed integers cannot be passed to unsigned integer variables of the same type without explicit conversion.

To make the usage of DynaXDF less complicated most function parameters which would normally declared as unsigned integer, e.g. XDF object handles, are declared as signed integer to get rid of permanent explicit data type conversions.

The usage of DynaXDF with C# is nearly identical in comparison to C++. The interface does generally not use events like in VB .Net because callback functions work very well in C#.

Data types in C#

All structures and enums used by DynaXDF are declared in the namespace DynaXDF. Because it is not possible to declare constants in a namespace, such constants are declared in the class CXDF. All data types, structures, and constants are defined in the file CXDF.cs. No further files are required to use DynaXDF.

Example (Visual C#):

```
using System;
using DynaXDF;
using System.Runtime.InteropServices.Marshal;

namespace hello_world
{
   class Hello_World
   {
      // Error callback function.
      static int XDFError(IntPtr Data, int ErrCode, IntPtr ErrMessage,
```

```
      int ErrType)
      {
         // The error type is a bitmask.
         Console.Write("{0}\n", PtrToStringAnsi(ErrMessage));
         Console.Write("\n");
         return 0; // We try to continue if an error occurs.
      }

      [STAThread]
      static void Main(string[] args)
      {
         try
         {
            String outFile = "c:/c#out.XDF";
            CXDF XDF = new CXDF();
            // Error messages are passed to the callback function.
            XDF.SetOnErrorProc(IntPtr.Zero, new DynaXDF.TErrorProc(XDFError));
            // We open the output file later if no error occurs.
            XDF.CreateNewXDF(null);
            // We use top down coordinates in this example
            XDF.SetPageCoords(DynaXDF.TPageCoord.pcTopDown);
            // Add an empty page to the file
            XDF.Append();
            // Before printing text you must set a font
            XDF.SetFont("Arial", DynaXDF.TFStyle.fsItalic, 20, true, DynaXDF.TCodepage.cp1252);
            XDF.WriteText(50.0, 50.0, "My first XDF output...");
            XDF.WriteText(50.0, 80.0, "File created: " + DateTime.Now.ToString());
            XDF.EndPage(); // Close the open page

            // No fatal error occurred?
            if (XDF.HaveOpenDoc())
            {
               // OK, now we can open the output file.
               if (!XDF.OpenOutputFile(outFile))
               {
                  // An error message was already passed to the error callback function
                  Console.Read();
                  return;
               }
               if (XDF.CloseFile())
               {
                  Console.Write("XDF file \"{0}\" successfully created!\n", outFile);
               }
            }
            XDF = null;
         }catch(Exception e)
         {
            Console.Write(e.Message);
         }
         Console.Read();
      }
   }
}
```

Embarcadero Delphi

The usage of DynaXDF with Embarcadero's Delphi is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class TXDF to make the usage easier. The instance pointer IXDF that is used by every DynaXDF function is hidden for the user in Delphi. The instance pointer is controlled by the wrapper class so that you don't need to create XDF instances manually.

To use DynaXDF with Delphi, proceed as follows:

- Add the interface file ***/include/Delphi/dynaXDF.pas*** to your project.
- Add the unit dynaXDF in the uses section in every source file where you want to use DynaXDF.
- Copy the ***dynaXDF.dll*** into a Windows search path (e.g. Windows/System32) or into your application directory, finished!

64 Bit Applications

Since Rad Studio XE2 you can develop 32 bit and 64 bit applications with Delphi. One thing that is often misunderstood is where the 32 bit and 64 bit versions of dynaXDF.dll must be stored. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynaXDF.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Delphi is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Usage

The Delphi interface encapsulates all DLL functions in the wrapper class TXDF. This class can be used like any other VCL class. The class is thread-safe and can be used without synchronization in multithreading applications.

However, some details must be known about the class. When the first instance is created, the constructor loads the dynaXDF.dll with the API function LoadLibrary(). When creating a further instance of the wrapper class TXDF, also a new XDF instance is created inside the DLL. Each instance of the wrapper class uses its own DLL instance.

If an instance of the wrapper class TXDF is destroyed, the destructor deletes the used XDF instance; if no other instance uses the library then it will be unloaded with the API function FreeLibrary().

However, the DLL is unloaded each time if the reference count of the DLL is zero. In most cases it makes sense to hold one instance of the wrapper class in memory to avoid unloading the library. The internal resources used by DynaXDF are always freed when CloseFile() is called (except when the file is created in memory), so that there is no need to destroy the main instance of TXDF.

Exception handling in Delphi

DynaXDF itself uses no native Delphi exception handling. Error messages and warnings are passed to an error callback function if any (see SetOnErrorProc()). If no callback function is used, then use the function GetErrorMessage() to get information about the last error.

However, the wrapper class TXDF uses native exceptions in the following cases:

- When creating a new instance of the wrapper class TXDF.
- When loading a DLL function with the API function GetProcAddress() (**all functions**).

If a function listed above fails, then an exception is raised by the class TXDF. Always encapsulate all function calls into a try / except block. Only a few exceptions can occur but these exceptions must not be ignored. Especially when using DynaXDF in multi-threading applications it is highly recommended to use try / except or try / finally blocks. A thread must always catch all exceptions inside the thread.

Using DynaXDF in Multithreading Applications

The usage of DynaXDF inside a thread is the same as in single-threaded applications.

However, if a callback function should be used, you must make sure that the callback function is declared in the same thread or that each thread uses its own copy of the callback function. In addition, it is highly recommended that only thread-safe functions are called inside the callback function. If any unsafe function must be executed the function that causes the execution of the callback function must be synchronized because it is impossible to synchronize a callback function itself.

Threads should be used completely isolated from the main-thread of the application. Function calls to and from the main-thread must be synchronized. The entire XDF file should be created inside the thread including the instance of the wrapper class TXDF. The class instance must also be deleted before the thread is terminated.

A running thread can be terminated at any time but it is highly recommended to wait for any running functions to end before a thread will be terminated. This can be done easily by checking the property Terminated within the thread before a new function is executed.

After a running function returns, the class instance can be destroyed by using the Free() method for that instance. This will clean up the used resources and the thread can be terminated. The instance of the wrapper class TXDF can be safely destroyed at any time after a running function

returned. All internal used resources will be freed, there is no need to call FreeXDF() manually beforehand.

Example (Single threaded):

In the following example we use a simple message box inside the error callback function. However, in a larger project it makes sense to output error messages into an error log or list box. DynaXDF ignores non-fatal errors by default so that it is possible to continue, but you can protocol each warning and errors during XDF creation.

```
unit Unit1;

interface

uses Windows, Messages, SysUtils, Classes, Controls, Forms, Dialogs,
StdCtrls, dynaXDF; // Include the file dynaXDF.pas to the unit
type
  TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
 private
 public
end;
var Form1: TForm1;
implementation
// First, we define our callback function that is called if an
// error occurred. Note: The calling convention is stdcall!
function ErrProc(const Data: Pointer; ErrCode: Integer; const
ErrMessage: PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
   s := Format('%s'#13'Abort processing?', [ErrMessage]);
   if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
      Result := -1 // break processing
   else
      Result := 0; // try to continue
end;
procedure TForm1.Button1Click(Sender: TObject);
var XDF: TXDF;
begin
   XDF := nil;
 try
   XDF := TXDF.Create;
   // set the error callback function first
   XDF.SetOnErrorProc(nil, @ErrProc);
   XDF.SetDocInfoA(diAuthor, 'Jens Boschulte');
   XDF.SetDocInfoA(diCreator, 'Delphi sample project');
   XDF.SetDocInfoA(diSubject, 'My first XDF file...');
```

```
    XDF.SetDocInfoA(diTitle, 'My first Delphi XDF output');
    XDF.SetViewerPreferences(vpDisplayDocTitle, avNone);

    XDF.CreateNewXDFA('c:\dout.XDF');
    XDF.Append;
    XDF.SetFontA('Arial', fsItalic, 40, true, cp1252);
    XDF.WriteFTextA(taCenter, 'My first Delphi output!!!');
    XDF.EndPage;
    XDF.CloseFile;
  except
    on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
  end;
  if XDF <> nil then XDF.Free;
end;
```

Compiling DynaXDF on Linux / UNIX

The build process of DynaXDF was designed to enable the compilation on most Linux or UNIX operating systems as easy as possible. All Linux and UNIX versions of DynaXDF can be compiled with GCC 4.2 or higher. GCC is freely available for most machine types.

> The pre-compiled Unix libraries are usually compiled with the default compiler that is common for the specific system, e.g. Solaris Studio on Solaris or Visual Age on AIX.

The configuration files are mainly designed for use with GCC and autoconf. If you want to use another compiler you must maybe change the compiler flags in the file configure.in and rebuild the configure script with autoconf. This can be done on an arbitrary Linux machine if no working autoconf is installed on the Unix machine.

If you get compilation issues just ask for a solution! We have a lot of experience with Unix systems, maybe we can help.

System requirements:

1. Properly installed GCC (4.2 or higher) C and C++ compiler. We strongly recommended GCC 4.2 or higher!
2. GNU make
3. To create a static library of DynaXDF you need also ar and ranlib

Build process

1. Copy first the entire directory dynaXDFent to your Linux or UNIX machine.
2. Change the access permissions of the following files as follows (subdirectory /source):
    a. chmod 777 config.guess
    b. chmod 777 config.sub
    c. chmod 777 confrel
    d. chmod 777 install-sh
3. Type "./confrel" and press enter. This command creates the make files for your machine and starts the compilation.
4. Clean up the directory with "make clean", finished!

Make install creates a static and shared library of DynaXDF and copies the libraries and header files, which are required to bind DynaXDF, into the subdirectory /source.

You find the following files in the subdirectory /source after compiling DynaXDF:

- `dynaXDF.h` `// Main header file of DynaXDF`
- `drv_conf.h` `// Required configuration file`
- `libdynaXDF.a` `// Static library`
- `libdynaXDF.so` `// Extension ".sl" on HP-UX or ".dylib" on Mac OS X`

Changing the configuration scripts

DynaXDF uses the freely available tool autoconf to create the main configuration script configure. Autoconf requires the file configure.in as input file which is located in the subdirectory /source. The final configure script can be executed without changes on all supported Linux and UNIX operating systems as well as on macOS. It creates the make files from the input files makefile.in which are located in all library directories; these files can normally be left unchanged. The top level makefile.in, which is stored in the subdirectory /source too, can be modified if further installation scripts should be executed after the library was successfully compiled. Note that the makefile.in files can be modified without rebuilding the configure script.

If you want to change certain compiler settings, or the compiler itself, modify the file configure.in and execute autoconf without parameters. Autoconf will rebuild the configure script with the new settings.

Linker flags

The used linker flags are designed to create a library with minimal dependencies so that DynaXDF can be delivered without other OS specific libraries. Depending on the target OS the linker flags can be changed so that OS specific libraries can be bind dynamically. This results in a smaller library but with more dependencies. To change the linker flags, modify the variable LD_LIBS in the file configure.in and rebuild the configure script with autoconf. The library can also be build LSB 3.0 compatible on Linux.

Compiler flags

When compiling DynaXDF on HP-UX the flag -fPIC (Position Independent Code) must be set at the minimum to enable the usage as shared library.

Optimization Level

DynaXDF is compiled with optimization level 3. This level is a good compromise between stable and fast code. The highest optimization level 4 causes a very long compilation time and it is possible that the resulting code is less stabile. Test the library properly before using this optimization level by default.

However, a release build should use the optimization level 3 or 4 because certain dependencies to internal GCC specific libraries are only removed if the optimization level is higher than 2.

Recommended compiler version

All Linux and Unix versions can be compiled with GCC 4.2. Older versions are not tested and maybe don't work. The GCC compiler should be configured with enabled POSIX compatible thread handling if possible, although the library does not depend on it.