DynaPDF 4.0

Reference & Manual

API Reference Version 4.0.37
January 8, 2020

Table of Contents

Legal Notices

Copyright: © 2003-2019 Jens Boschulte, DynaForms GmbH. All rights reserved.
DynaForms GmbH
Buchecker Street 26
D-58256 Gießen, Germany
Trade Register HRA 9775, District Court Gießen
CEO Jens Boschulte
Phone: +49 26 32-466 79 37
Fax: +49 26 32-466 79 38

Table of Contents

Digital Signatures

Signature functions

What is stored in a signature field?

How to create a signature field?

Who can create a signature field?

What kind of signature fields are supported?

Field properties

Field names

Accessing

Table Functions

AddColumn

AddRows

AddRowsToTable

AddTable

ClearTable

DeleteRows

DeleteTable

DrawTable

GetFirstRow

GetFlags

GetNumCols

GetNumRows

InsertColumn

InsertRows

ModifyField

PrepareTable

RemoveColumn

RemoveRowsFromTable

SetFlag

SetField

SetSignature

SetSignatureValue

SetTable

SetTableColor

SetTableColumn

SetTableColumnProperties

SetTableFont

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

SetTableFontMatrix

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

### Var Parameters

```c
#define ADDR &
```

Functions in DynaPDF, which pass a value to a function parameter are handled differently in C and C++ as does not support the address operator & so that our parameters are defined as normal pointers to pointers in C. DynaPDF 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 DynaPDF 2.5 all structures which can be extended in future versions contain the member StructureSize. This variable must be set to size (StructureSize). 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**

```c
... 
const myStruct = AddressOf(myStruct);
```

### Multi-byte Strings

#### Unicode

DynaPDF 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 (IE or BE) as default string format such as Linux or most UNIX OS, all strings must be converted to UTF-16 before passing to DynaPDF.

You can use the predefined macro ToUTF16 to do this.

#### ToUTF16

```c
#define ToUTF16(IPDF, s)( pdfUTF32ToUTF16 ((IPDF), ( UI32 *)(s)) )
```

This macro calls pdfUTF32ToUTF16() only if the OS uses UTF-32 as Unicode string format.

- **UTF-16**
  -UTF-16 strings are available. If you need to store more than 6 string variables then you must copy the converted strings to 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 DynaPDF 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

CK multi-byte strings contain mixed shift JIS/shift 30-bit character codes. A CK string can be defined as an ANSI string (data type char*) and as multi-byte string (data type UI16*). The multi-byte format for CJK multi-byte strings is UTF-16. DynaPDF supports UTF-16 in little-endian and big-endian mode. DynaPDF converts the string members of a structure into UTF-16 if the structure size is set to 2.5 or higher.
data types  Page 17 of 777

Exception handling  Page 18 of 777

Exception handling  Page 19 of 777

Exception handling  Page 20 of 777

Exception handling  Page 21 of 777

Exception handling  Page 38 of 777

Exception handling  Page 39 of 777

Exception handling  Page 40 of 777

Usage of the event functions is quite easy; just declare a local instance variable of the wrapper

class CPDF in the Option Explicit section of the unit as follows:

```delphi
Private  FRunning As Boolean
```

The usage of events in Visual Basic or VB .Net is exactly the same as in VB 6.0.

Data types

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.

<table>
<thead>
<tr>
<th>C type</th>
<th>C++ type</th>
<th>Delphi type</th>
<th>VB type</th>
<th>VB .Net type</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char</td>
<td>WideString</td>
<td>String</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>_U8</td>
<td>_U8</td>
<td>Pointer</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>_U32</td>
<td>_Int32</td>
<td>Integer</td>
<td>Integer</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td></td>
</tr>
<tr>
<td>_BSTR</td>
<td>WideString</td>
<td>Pointer</td>
<td>WideString</td>
<td>WideString</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>void</td>
<td>Pointer</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>PDF_CALL</td>
<td>__stdcall</td>
<td>Windows only, otherwise empty</td>
<td>Windows only, otherwise empty</td>
<td>Windows only, otherwise empty</td>
<td></td>
</tr>
</tbody>
</table>

The right combo box contains the available events, when selecting the event "Error" VB adds

automatically an empty event procedure to your source code:

There are 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
characters sets (_mpg8, _mpg16, _mpg32 and so on), see 7.2.10) 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, DynaPDF must convert the incoming CJK string to
Transformation-look for it in the supported 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.

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
characters sets (_mpg8, _mpg16, _mpg32 and so on), see 7.2.10) 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, DynaPDF must convert the incoming CJK string to
Transformation-look for it in the supported 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.

<table>
<thead>
<tr>
<th>C type</th>
<th>C++ type</th>
<th>Delphi type</th>
<th>VB type</th>
<th>VB .Net type</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char</td>
<td>WideString</td>
<td>String</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>_U8</td>
<td>_U8</td>
<td>Pointer</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>_U32</td>
<td>_Int32</td>
<td>Integer</td>
<td>Integer</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td></td>
</tr>
<tr>
<td>_BSTR</td>
<td>WideString</td>
<td>Pointer</td>
<td>WideString</td>
<td>WideString</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>void</td>
<td>Pointer</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>PDF_CALL</td>
<td>__stdcall</td>
<td>Windows only, otherwise empty</td>
<td>Windows only, otherwise empty</td>
<td>Windows only, otherwise empty</td>
<td></td>
</tr>
</tbody>
</table>

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

which 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

DynaPDF 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 closed function. When DynaPDF tries to close

an event inside the closed function an access violation occurs and VB crashes. VB.Net does not

crash but raises a System.Threading.ThreadStateException in that case.

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

Option Explicit

Private WithEvents PDF As DynaPDF
End Sub

Private Sub Command1_Click()
If FRunning Then Exit Sub 'Check whether a previous call is running
FRunning = True 'Call new DynaPDF functions here...
DoEvents 'Process user 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.
### Custom Library Changes

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

#### Compiler Switches

DynaPDF supports several compiler switches to disable unnecessary features. The following macros disable or enable one of the image libraries used by DynaPDF as well as other features.

- `#define DRV_SUPPORT_PSD 1 // about 1 KB` for enabling PSD support.
- `#define DRV_SUPPORT_PNG 1 // about 100 KB` for enabling PNG support.
- `#define DRV_SUPPORT_PGM 1 // PBM, PGM, PNM, PPM Image formats, ~1 KB` for enabling PGM support.
- `#define DRV_SUPPORT_TIFF 1 // about 310 KB` for enabling TIFF support.
- `#define DRV_SUPPORT_RC4 1 // RC4 encryption and decryption` for enabling RC4 encryption.
- `#define DRV_SUPPORT_IMP 1  // about 50 KB (PDF import)` for enabling PDF import support.
- `#define DRV_SUPPORT_REDESIGN 1 // about 170 KB` for enabling a redesign of some features.

### Customized Exception handling

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

- `#define DYNPDF_EXCEPTION_FATAL 0x10000000` for fatal errors.
- `#define DYNPDF_EXCEPTION_ERROR 0x00000008` for errors.
- `#define DYNPDF_EXCEPTION_WARNING 0x00000004` for warnings.
- `#define DYNPDF_EXCEPTION_VALUE_ERROR 0x00000002` for value errors.
- `#define DYNPDF_EXCEPTION_SYNTAX_ERROR 0x00000001` for syntax errors.
- `#define DYNPDF_EXCEPTION_ALL_ERRORS 0x0000FFFF` for all errors.
- `#define DYNPDF_EXCEPTION_IGNORE_ALL 0x00000000` for ignoring all errors.

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 DynaPDF 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 recommend that you debug especially the function `CreateNewPDF()`. All objects must be prepared for writing in a two stage phase to preserve object numbers. The first stage assigns 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 PDF file is described in memory as a 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 no exception occurs if an object must be deleted:

1. The owner of all resources and global objects is `CObj`. 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 `PDF::PreparePage(Obj)` if the object is part of a page. If the object is not included in a page then `CreateObject()` must be called in `CObj::PrepareObject()`.
3. All classes which bold pointers to other object classes must check the “Used” flag before writing the object data to file (`GetUnUsed()` is a number 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, and so on must never be deleted at runtime and their “Used” flag must NEVER be unset.
7. All resource objects must be derived from `CBaseResources`.
8. Object classes must set the “Used” flag of resource object, if the class is used by this object.
9. The “Used” flag of two-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 deleted 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 PDF file will be damaged.
12. Make sure that the function `FreePDF()` can be called at any time without causing memory leaks or other unwanted side effects.

#### Requirements to add your own code to DynaPDF

If you want to make your own feature 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 already used by DynaPDF (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 `pdf_utils.h` before using an external function or template. The implementations used by DynaPDF 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. DynaPDF uses already a few GDI functions to convert WMF files to EMF and 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 DynaPDF class `CString` or `CPDFString` instead. These classes support many PDF specific functions.

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

To use DynaPDF 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\dynapdf.h` into the units which will use DynaPDF functions.
3. Add the import library `implib_dynapdf.lib` to your project.
4. Copy the `dynapdf.dll` into a Windows search path (e.g., Windows\System32) or into the output directory.
5. Finished!

If you want to use DynaPDF with a C++ project, add the line “using namespace DynaPDF;” after including the header file `dynapdf.h`.

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

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

If you want to use DynaPDF with an older version of C++ Builder, you may rebuild the libraries and the `dynapdf.dll` into the same directory and execute the command above.

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

```
implib dynapdf.dll dynapdf.lib
```

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

Example (C++ Builder):

```
#include <vcl.h>

Example (C++ Builder):

```

Language bindings

Differences between DynaPDF interfaces

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

All DynaPDF functions contain an instance pointer of the active PDF instance (constant `PDF*` 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 PDF instance automatically.

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

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

Visual Basic, Visual C# or Delphi users must create an instance of the wrapper class `CTPDF` or `TPDF` in Delphi before a DynaPDF 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 PDF instance with the function `pdfNewPDF()` before a DynaPDF function can be executed. This instance must be deleted with the function `pdfDeletePDF()` when it is no longer needed.

We tried also to consider the specific requirements for each programming language so that DynaPDF 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.

Language Bindings

Differences between DynaPDF interfaces

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

All DynaPDF functions contain an instance pointer of the active PDF instance (constant `PDF*` 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 PDF instance automatically.

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

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

Visual Basic, Visual C# or Delphi users must create an instance of the wrapper class `CTPDF` or `TPDF` in Delphi before a DynaPDF 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 PDF instance with the function `pdfNewPDF()` before a DynaPDF function can be executed. This instance must be deleted with the function `pdfDeletePDF()` when it is no longer needed.

We tried also to consider the specific requirements for each programming language so that DynaPDF 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.

Language Bindings

Differences between DynaPDF interfaces

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

All DynaPDF functions contain an instance pointer of the active PDF instance (constant `PDF*` 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 PDF instance automatically.

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

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

Visual Basic, Visual C# or Delphi users must create an instance of the wrapper class `CTPDF` or `TPDF` in Delphi before a DynaPDF 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 PDF instance with the function `pdfNewPDF()` before a DynaPDF function can be executed. This instance must be deleted with the function `pdfDeletePDF()` when it is no longer needed.

We tried also to consider the specific requirements for each programming language so that DynaPDF 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.
Microsoft Visual C++

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

1. Open a new project, or your favourite project, in Visual Studio.
2. Add the header file `include/C_CPP/dynapdf.h` to your project (menu Project/Add Class Module/Existing...).
3. Add the file `include/C_CPP/dynapdf.cpp` to your project (menu Project/Add Class Module/Existing...).
4. If you want to use the table class then add also the file `include/C_CPP/dynapdfTab.h` to your project (menu Project/Add Class Module/Existing...).
5. Finally, make sure that the DynaPDF library is found by the compiler. This is done automatically by the wrapper class so that you don't need to create PDF instances manually.

To use DynaPDF with Visual Basic proceed as follows:

1. Open a new project, or your favourite project, in Visual Studio.
2. Include the header file `include/Visual_Basic/CPDF.cls` into your project (menu Project/Add Class Module/Existing...).
3. Add the file `include/Visual_Basic/dynapdf.dll` into your project (menu Project/Add Module/Existing...).
4. Add the file `include/Visual_Basic/dynapdflib.exe` to your project (menu Project/Add Class Module/Existing...).

The usage of DynaPDF with Visual Basic is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF which is used by every DynaPDF function is hidden for the user in the wrapper class. This has the advantage that you don't need to create PDF instances manually.

The usage of DynaPDF with Visual Basic 6.0 is essentially the same as in C++, with the exception that the namespace DynaPDF must be declared.

A PDF instance can be created in an arbitrary context of C++ files. All used resources are automatically freed when the PDF file is closed (except when the PDF file was created in memory).

To improve processing speed, use one instance as long as possible.

Most examples are written in C or C++ which can directly be used with an EmbraCAD or Microsoft Compiler.

Example C++:

```cpp
#include <Dynapdf.h>

using namespace DynaPDF;

int main(int argc, char **argv)
{
    // Open a file for a new PDF instance below.
    dynapdf::pdf = pdfNewPDF();
    // Create a new PDF instance

dynaPDF was compiled and developed with Microsoft Visual Studio 6.0 SP 6 and Visual Studio 2005. Two versions of the library are delivered:

• The library `dynapdf.dll` is used with the SafeInited DLL.
• The library `dynapdfm.dll` is used with the Multithreaded DLL.

Both versions are fully compatible to VC++ 7.0 or higher (Visual Studio .Net). To avoid conflicts with different standard library versions choose the right DynaPDF DLL and import library depending on your project settings. A pre-compiled single threaded version of DynaPDF is also available.

Please note that the dynapdf.dll does not support PDF files larger than 2 GByte. When the library is compiled with Visual Studio 2005 or higher, this limitation does no longer exist. However, in this case, you must define the Visual Studio Runtime library MYCYRCHR.DLL or higher with your application. This library causes other problems since it must be installed on the system and cannot be placed in the application folder.

The right combobox contains the available events. When selecting the event "Error" VB adds automatically an empty event procedure to your source code:

Private Sub __Dynapdf_Error(ByVal Description As String, ByVal ErrType As Long, ByVal ErrMessage As String)
    ' DoBreak to True
    ' DoBreak to False
End Sub

If possible, then use the dynapdf.dll instead. This library requires no additional dependencies to enable 64 bit file support.

If you want to use DynaPDF with a C++ project, add the line "#include <Dynapdf.h>" after including the header file `dynapdf.h`.

The usage of DynaPDF with Visual Basic is essentially the same as in C++, with the exception that the namespace DynaPDF must be declared.

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

Most examples are written in C or C++ which can directly be used with an EmbraCAD or Microsoft Compiler.

Example C++:

```cpp
#include <Dynapdf.h>

using namespace DynaPDF;

int main(int argc, char **argv)
{
    // Open a file for a new PDF instance below.
    dynapdf::pdf = pdfNewPDF();
    // Create a new PDF instance

    pdfWriteFText(pdf, taCenter, "My first C++ output!");
    // Set the error callback first
    pdfSetErrorCallback(pdf, SI32  PDF_CALL PDFError( const void * Data, SI32  ErrCode, const char * Description, char * ErrorMessage));

    // Call some DynaPDF functions...
    pdfSetFont(pdf, "Arial", fsBold, 40.0, true, cp1252);
    pdfCreateNewPDF(pdf, "c:/myfirst.pdf");
    pdfDeletePDF(pdf); // Do not forget to delete the PDF instance

    return 0;
}
```

The functions are not exported, they do not conflict with the original declarations. These

The standard exception handling of DynaPDF 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 DynaPDF API function calls.

To enable the event support of the wrapper class CPDF declare a local instance variable as

```cpp
Private m_abCPDF As CPDF "global instance variable"
```

The instance variable PDF is now listed in the left combobox of the VB code editor.

The usage of DynaPDF with Visual Basic is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF which is used by every DynaPDF function is hidden for the user in the wrapper class. This has the advantage that you don't need to create PDF instances manually.

To use DynaPDF with Visual Basic proceed as follows:

1. Open a new project, or your favourite project, in Visual Studio.
2. Include the header file `include/C_CPP/dynapdf.h` into your project (menu Project/Add Class Module/Existing...).
3. Add the file `include/C_CPP/dynapdflib.exe` to your project (menu Project/Add Class Module/Existing...).
4. Add the file `include/Visual_Basic/CPDF.cls` into your project (menu Project/Add Class Module/Existing...).

Due to implementation limits of Microsoft's Intellisense, code completion does not work with this file. However, it seems that this issue was partially solved with Update Pack 6 of Microsoft's Visual Studio. If code completion does not work in your environment then include this file!


The usage of DynaPDF with Visual Basic is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF which is used by every DynaPDF function is hidden for the user in the wrapper class. This has the advantage that you don't need to create PDF instances manually.

The usage of DynaPDF with Visual Basic 6.0 is essentially the same as in C++, with the exception that the namespace DynaPDF must be declared.

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

Most examples are written in C or C++ which can directly be used with an EmbraCAD or Microsoft Compiler.

Example C++:

```cpp
#include <Dynapdf.h>

using namespace DynaPDF;

int main(int argc, char **argv)
{
    // Open a file for a new PDF instance below.
    dynapdf::pdf = pdfNewPDF();
    // Create a new PDF instance

    pdfWriteFText(pdf, taCenter, "My first C++ output!");
    // Set the error callback first
    pdfSetErrorCallback(pdf, SI32  PDF_CALL PDFError( const void * Data, SI32  ErrCode, const char * Description, char * ErrorMessage));

    // Call some DynaPDF functions...
    pdfSetFont(pdf, "Arial", fsBold, 40.0, true, cp1252);
    pdfCreateNewPDF(pdf, "c:/myfirst.pdf");
    pdfDeletePDF(pdf); // Do not forget to delete the PDF instance

    return 0;
}
```

If possible, then use the dynapdf.dll instead. This library requires no additional dependencies to enable 64 bit file support.

If you want to use DynaPDF with a C++ project, add the line "#include <Dynapdf.h>" after including the header file `dynapdf.h`.

The usage of DynaPDF with Visual Basic is essentially the same as in C++, with the exception that the namespace DynaPDF must be declared.

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

Most examples are written in C or C++ which can directly be used with an EmbraCAD or Microsoft Compiler.

Example C++:

```cpp
#include <Dynapdf.h>

using namespace DynaPDF;

int main(int argc, char **argv)
{
    // Open a file for a new PDF instance below.
    dynapdf::pdf = pdfNewPDF();
    // Create a new PDF instance

    pdfWriteFText(pdf, taCenter, "My first C++ output!");
    // Set the error callback first
    pdfSetErrorCallback(pdf, SI32  PDF_CALL PDFError( const void * Data, SI32  ErrCode, const char * Description, char * ErrorMessage));

    // Call some DynaPDF functions...
    pdfSetFont(pdf, "Arial", fsBold, 40.0, true, cp1252);
    pdfCreateNewPDF(pdf, "c:/myfirst.pdf");
    pdfDeletePDF(pdf); // Do not forget to delete the PDF instance

    return 0;
}
```

The error block cannot be executed because DynaPDF 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 is that this error block does not affect the normal exception handling of DynaPDF.

The DoEvents procedure

The usage of events in Visual Basic is quite easy but there is a special behavior 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
The usage of DynaPDF 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 CPDF to make the usage easier. The instance pointer CPDF that is used by every DynaPDF 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 PDF instances manually.

To use DynaPDF with VB.Net proceed as follows:

1. Add the file */include/Visual_Basic_Net/CPDF.vb* to your project (menu Project/Add Existing Element...).
2. Finally, make sure that the dynapdf.dll can be found; just copy the DLL into Windows/System32 directory!

**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 64 bit. This is impossible since you can either link the 32 bit dynapdf.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 dynapdf.dll. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynapdf.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 files in your project. Otherwise you must update the interface files manually whenever you install a newer version of DynaPDF.

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

---

**Exception handling in VB.Net**

The exception handling in Visual Basic.Net is the same as in Visual Basic 6.0. The class CPDF 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.

It is also possible to declare callback functions instead of events but the use of callback functions is rather complicated in VB.Net. While C# handles callback functions correctly without further considerations, VB.Net users must test their callback functions properly. It seems that VB.Net handles callback functions differently. When DynaPDF tries to execute the callback function it must make sure that the callback function cannot be executed again while a previous call of the function is still running. 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. VB.Net applications. If you cannot find a data type in the class CPDF then take a look into the file DynaPDFInt.vb.

**Data types used by DynaPDF**

DynaPDF uses a large set of enums and other data types which are mostly declared within the class CPDF. However, a few data types are declared in the file DynaPDFInt.vb. This must be taken into account when developing VB.Net applications. If you cannot find a data type in the class CPDF then take a look into the file DynaPDFInt.vb.

**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 DynaPDF 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 DynaPDF tries to raise an event inside of the cloned function an access violation occurs and VB crashes.

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

```vbnet
Option Explicit
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As Long, DoBreak As Boolean)
    DoBreak = (MsgBox(Description, vbExclamation Or vbYesNo, "Error") = vbYes)
End Sub
```

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

---

**Language Bindings  Page 35 of 777**

Visual Basic.Net

The usage of DynaPDF 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 CPDF to make the usage easier. The instance pointer CPDF that is used by every DynaPDF 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 PDF instances manually.

To use DynaPDF with VB.Net proceed as follows:

1. Add the file */include/Visual_Basic_Net/CPDF.vb* to your project (menu Project/Add Existing Element...).
2. Finally, make sure that the dynapdf.dll can be found; just copy the DLL into Windows/System32 directory!

**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 64 bit. This is impossible since you can either link the 32 bit dynapdf.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 dynapdf.dll. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynapdf.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 files in your project. Otherwise you must update the interface files manually whenever you install a newer version of DynaPDF.

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

**Exception handling in VB.Net**

The exception handling in Visual Basic.Net is the same as in Visual Basic 6.0. The class CPDF 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.

It is also possible to declare callback functions instead of events but the use of callback functions is rather complicated in VB.Net. While C# handles callback functions correctly without further considerations, VB.Net users must test their callback functions properly. It seems that VB.Net handles callback functions differently. When DynaPDF tries to execute the callback function it must make sure that the callback function cannot be executed again while a previous call of the function is still running. 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. VB.Net applications. If you cannot find a data type in the class CPDF then take a look into the file DynaPDFInt.vb.

**Data types used by DynaPDF**

DynaPDF uses a large set of enums and other data types which are mostly declared within the class CPDF. However, a few data types are declared in the file DynaPDFInt.vb. This must be taken into account when developing VB.Net applications. If you cannot find a data type in the class CPDF then take a look into the file DynaPDFInt.vb.

**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 DynaPDF 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 DynaPDF tries to raise an event inside of the cloned function an access violation occurs and VB crashes.

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

```vbnet
Option Explicit
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As Long, DoBreak As Boolean)
    DoBreak = (MsgBox(Description, vbExclamation Or vbYesNo, "Error") = vbYes)
End Sub
```
However, when using Delphi it is possible that a user clicks on the button again that executes DynaPDF functions while a previous call is still running. This is a normal problem but when using events the event function becomes invalid. This is the same behavior as in Visual Basic 6.0 with the exception that: Net does not crash, a SystemErrorException exception 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++ or Delphi will never cause an access violation or exception here.

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

```vbnet
Private FRunning As Boolean
Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    FRunning = False
    DoEvents 'Process messages
    FRunning = True
    If FRunning Then Exit Sub 'Check whether a previous call is running
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 stable.

Example (Visual Basic .Net):

```vbnet
Private WithEvents FPDF As CPDF 'Enable event support

Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    FRunning = False
    DoEvents 'Process messages
    FRunning = True
    If FRunning Then Exit Sub 'Check whether a previous call is running
End Sub
```

Use DynaPDF with Visual C# proceed as follows:

- Add the file `/include/Visual_C#/CPDF.cs` to your project (menu Project/Add Existing Item…).
- Make sure that the `dynapdf.dll` can be found, just copy the DLL into the Windows/System32 directory. Included!

### 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 be set to Desktop. This is impossible since you can either link the 32 bit `dynapdf.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 `dynapdf.dll`. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the `dynapdf.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 find it. The installer of your application should copy the DLL into the application directory instead.

### General Note:

Visual Studio not copies the interface file CPDF.cs 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 files to your project. Otherwise you must update the interface manually whenever you install a newer version of DynaPDF.

All DynaPDF functions are encapsulated in the wrapper class CPDF. This class makes sure that the DynaPDF functions can be used without limitations and programming with DynaPDF 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.

### Visual C#

The usage of DynaPDF with Visual C# is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier.

The instance pointer IPDF that is used by every DynaPDF function is hidden for the user in C#.

The instance pointer is controlled by the wrapper class so that you must not create PDF instances manually.

To use DynaPDF with Visual C# proceed as follows:

- Add the file `/include/Visual_C#/CPDF.cs` to your project (menu Project/Add Existing Item…).
- Make sure that the `dynapdf.dll` can be found, just copy the DLL into the Windows/System32 directory. Included!

### Data types in C#

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

Example (Visual C#)

```csharp
using System;
using DynaPDF;

namespace hello_world
{
    class Hello_World
    {
        // Error callback function.
        static int PDFError(IntPtr Data, int ErrorCode, IntPtr ErrorMessage, int ErrorType)
        {
            // The error type is a byte
            Console.WriteLine("Error: ", Marshal.PtrToStringAuto(ErrorMessage));
            Console.WriteLine("Error: ");
            return 0; // We try to continue if an error occurs.
        }

        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                String outFile = "c:/c#out.pdf";
                PFDVFCreateNewPDFA("c:/vbout.pdf")
                FPDF.Append("Hello, World!
               ")
                FPDF.EndPage()
                FPDF.CloseFile()
            }
            catch (DynaPDFException e)
            {
                if (e.Type == (DynaPDF.TErrorProc(PDFError)))
                {
                    Error = "PDF Error: 
                    " + Marshal.PtrToStringAuto(e.Message);
                }
                throw new Exception("A DynaPDF Error has occurred.
               " + Error);
            }
        }
    }
}
```
However, the DLL is unloaded each time if the reference count of the DLL is zero. In most cases it makes sense to load one instance of the wrapper class in memory to avoid unloading the library. The internal resources used by DynaPDF 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 TPDF.

### Exception handling in Delphi

DynaPDF 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.

For example:

```delphi
function GetErrorMessage(ErrorMessage: PChar; ErrType: Integer): Integer;  
begin
  Result := 0; // try to continue
end;
```

In the following example we use a simple message box inside the error callback function.

```delphi
procedure Button1Click(Sender: TObject);
var
  Result: Integer;
begin
  // First, we define our callback function that is called if an error occurs.
  // The calling convention is cdecl.
  function ErrorProc(const errorData: Pointer; ErrorCode: Integer; ord(realerrName): PChar): Integer; cdecl;
  var
    s: String;
  begin
    s := Format("An error occurred: %s", [realerrName]);
    if ErrorMessage(e, errorData, ErrorCode, ErrorCode, s) then Result := -1 // break processing
  end;
  
  // load the library if necessary
  if FileExists('dynapdf.dll') then begin
    freeLibrary(FreeLibrary dynapdf.dll);
    Result := LoadLibrary(dynapdf.dll, Ord('dynapdf')), ERR_NAME_NOT_FOUND;
  end;

  if Result = nil then begin
    Result := -1; // break processing
  end
end;
```

### 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 dynapdf.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 dynapdf.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 TPDF. 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 dynapdf.dll with the API function LoadLibrary(). When creating a further instance of the wrapper class TPDF, also a new PDF instance is created inside the DLL. Each instance of the wrapper class uses its own DLL instance.

If an instance of the wrapper class TPDF is destroyed, the destructor deletes the used PDF instance if no other instance uses the library then it will be unloaded with the API function FreeLibrary().
supported Linux and UNIX operating systems as well as on Mac OS X. It creates the make files from the input files makefiles 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, may 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.

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

**Linker flags**

The used linker flags are designed to create a library with minimal dependencies so that DynaPDF 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 linked dynamically. This results in a smaller library but with more dependencies. To change the linker flags, modify the variable $LIB_LDR in the file configure.in and rebuild the configure script with autoconf.

**Compiler flags**

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

**Optimization Level**

DynaPDF 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 too stable. 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 in internal GCC specific libraries are only removed if the optimization level is higher than 2.

**Recommended compiler version**

Most DynaPDF versions for Linux and UNIX are compiled with GCC 4.2 or higher. Due to certain bugs in older versions of GCC, make sure that DynaPDF can be compiled at the minimum with GCC 5.0. Especially version 2.96 contains many bugs. If you use a precompiled library of DynaPDF, please note that DynaPDF is not binary compatible to GCC 2.96 or earlier. The GCC compiler should be configured with PTHREAD compatible thread handling if possible, although the library does not depend on it.

### Compiling DynaPDF on Linux / UNIX

PDF creation:

```delphi
pdf := TPDF.CreateNewPDF('c:\dout.pdf');
pdf.SetDocInfo('My first Delphi PDF output');
pdf.SetViewerPreferences(vpDisplayDocTitle, avNone);
pdf.WriteString('Hello World!', 'Hello', cp1252);
pdf.CloseFile;
```

Adding text to the PDF:

```delphi
pdf := TPDF.CreateNewPDF('c:\dout.pdf');
pdf.SetDocInfo('My first Delphi PDF output');
pdf.SetFont('Arial', fsBold, 40, true, cp1252);
pdf.WriteByteText('My first Delphi output!!!', taCenter);
pdf.WriteByteText('My second Delphi output!!!', taCenter);
pdf.Free;
end;
```

### Compiling DynaPDF on Mac OS X

PDF creation:

```delphi
pdf := TPDF.CreateNewPDF('c:\dout.pdf');
pdf.SetDocInfo('My first Delphi PDF output');
pdf.SetViewerPreferences(vpDisplayDocTitle, avNone);
pdf.WriteString('Hello World!', 'Hello', cp1252);
pdf.CloseFile;
```

Adding text to the PDF:

```delphi
pdf := TPDF.CreateNewPDF('c:\dout.pdf');
pdf.SetDocInfo('My first Delphi PDF output');
pdf.SetFont('Arial', fsBold, 40, true, cp1252);
pdf.WriteByteText('My first Delphi output!!!', taCenter);
pdf.WriteByteText('My second Delphi output!!!', taCenter);
pdf.Free;
end;
```

**System requirements**

1. Properly installed GCC (3.2 or higher) and C++ compiler. We strongly recommend GCC 4.0 or higher!
2. GNU make
3. To create a static library of DynaPDF you need also ar and ranlib

**Build process**

1. Copy the entire directory dynapdf_ent to your Linux or UNIX machine.
2. Change the access permissions of the following files as follows (subdirectory /source):
   - `chmod 777 configure.in`
   - `chmod 777 install.sh`
   - `chmod 777 install.sh`
3. Type `./configure` 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 DynaPDF and copies the libraries and header files, which are required to bind DynaPDF, into the subdirectory /source.

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

- `dynapdf.h` // Main header file of DynaPDF
- `dynapdf.a` // STATIC library
- `libdynapdf.a` // Extension `.sl` on HP-UX or `.dylib` on Mac OS X

**Changing the configuration scripts**

DynaPDF uses the freely available tool Autoconf to create the main configuration script configure. Autoreconf 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 platforms.
Interactive Forms

DynaPDF supports a set of functions to create and edit form fields held predefined actions and JavaScript actions. This section describes how an Interactive Form can be created and how certain features can be used.

Field Appearance

Interactive Form Fields support user defined background, text and border colors, as well as different border styles. These properties can be set or changed with following functions:

- Global properties for new created fields:
  - `Get/SetColorSpace()` // Color space
  - `Get/SetLineWidth()` // Line width of the border

- Functions to change the appearance of an existing field:
  - `SetFieldBBox()` // Changes the field's bounding box
  - `Get/SetFieldBorderColor()` // Border color
  - `Get/SetFieldBackgroundColor()` // Background color
  - `Get/SetFieldTextAlign()` // Text alignment

Global appearance properties:

- `Get/GetAppearance()` // Get appearance of a page
- `Get/GetFieldAppearance()` // Get appearance of a field

Calc order:

- `Set/GetFieldCalcOrder()` // Set or get the calc order

Other:

- `DeleteAllAppearance()` // Delete all appearances of a page
- `DeleteAppearance()` // Delete a specific appearance
- `DeleteAllActions()` // Delete all actions of a page

The global NeedAppearance flag of an Interactive Form defines whether the viewer should create the field appearances on demand when opening the file or whether the existing definitions should be taken from the PDF file. DynaPDF creates always appearance streams for all field types with exception of barcode fields. However, in certain cases it can be useful to let the viewer render fields with their own definitions because the exact way how Adobe's Acrobat builds the field appearances is not documented.

For example, when editing the contents of a text field in Adobe's Acrobat the viewer rebuilds the field appearance before placing the editing cursor into the field. The new appearance created from Adobe's Acrobat can be slightly different in comparison to the one that was created by DynaPDF. The visible content, especially of text fields, is sometimes not absolutely stable.

If the NeedAppearance flag is set, the viewer uses already its own algorithms to build the field appearances when opening the file. This avoids visible changes when editing a field. However, the NeedAppearance flag must not be set to true if a form contains page templates.

Important field properties when creating new fields

The line width of the border field is derived from the current graphics state when a new field is created (see `SetLineWidth()`). No border will be drawn if either the line width is set to zero or if the border color is set to NO_COLOR (see `SetFieldBorderColor()`). The default background color for new fields is NO_COLOR, that means the background appears transparent. Form fields support the color spaces DeviceGray, DeviceRGB, and DeviceCMYK. The default background, border, and text color must be defined in the current color space. Note that DynaPDF does not convert the current color values if the color space will be changed.

Field Properties

Most field properties and values can be read and changed with DynaPDF. The following list gives an overview over the available functions and for what they can be used:

- `Get/GetFieldExValCount()` // Number of values/ export values
  - `Get/GetFieldExpVal()` // Export value of a field
  - `Get/GetFieldMapName()` // Mapping name -> export name

How to change the tabulator order?

One important field property is the so called "Tabulator Order". The tabulator order can be changed at two different points:

1. When the file is created (see `SetFieldIndex()`)
2. When the file is opened (see `SetFieldTabulatorOrder()`)

Global field appearance properties:

- `Get/GetFieldAppearance()` // Get appearance of a field
  - `Get/GetUnknownAppeareance()` // Get appearance of a field

Calc order:

- `Set/GetFieldCalcOrder()` // Get or change the calc order

Other:

- `DeleteAllAppearance()` // Delete all appearances of a page
- `DeleteAppearance()` // Delete a specific appearance
- `DeleteAllActions()` // Delete all actions of a page

The global NeedAppearance flag of an Interactive Form defines whether the viewer should create the field appearances on demand when opening the file or whether the existing definitions should be taken from the PDF file. DynaPDF creates always appearance streams for all field types with exception of barcode fields. However, in certain cases it can be useful to let the viewer render fields with their own definitions because the exact way how Adobe's Acrobat builds the field appearances is not documented.

For example, when editing the contents of a text field in Adobe's Acrobat the viewer rebuilds the field appearance before placing the editing cursor into the field. The new appearance created from Adobe's Acrobat can be slightly different in comparison to the one that was created by DynaPDF. The visible content, especially of text fields, is sometimes not absolutely stable.

If the NeedAppearance flag is set, the viewer uses already its own algorithms to build the field appearances when opening the file. This avoids visible changes when editing a field. However, the NeedAppearance flag must not be set to true if a form contains page templates.

When changing a value or property of a Field Group there is nothing special that must be considered. DynaPDF sets the wished value or property automatically to the right field.

Example:

```java
... 
pt = pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
... pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
... pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
... pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
```
**Field Names**

Interactive Form Fields are identified over the field name in a viewer application. A field name is an Ansi string that should be human-readable. Beginning with PDF 1.5, field names can also be defined as Unicode strings. However, all functions to create new fields in DynaPDF support Ansi settings only. All characters within the Ansi character set (code page 1252) can be used with exception of the period character (.) and characters below index 32.

A field name should not end with a space character because Adobe's Acrobat is then sometimes unable to access such a field with a JavaScript Action or function.

The period (.) is a reserved character because it is used to build the fully qualified field name in a viewer application. The fully qualified field name is constructed from the partial field name of the field and all of its ancestors.

For a field with no parent group field, the partial and fully qualified names are the same. For a field that is the child of another field, the fully qualified name is formed by appending the child field's partial name to the parent's fully qualified name, separated by a period, e.g. Address.Street.

**Fields with identical names**

It is possible to create two or more fields of the same type which use the same name. Such fields contain always the same value if the value of one field of the group is changed.

Fields with identical names are internally represented as a special type of field group, which is automatically created by DynaPDF. This makes the handling more complicated because the children of such a group do not contain a field name. The name is set to the parent's group field but not to the children of the group. This can normally be ignored but when enumerating fields with GetField() or GetFormField() you must consider that not all fields contain a name, the parent field's handle is set instead.

However, with the exception described above, field names must be unique within the hierarchy in which they appear. This is especially important when multiple Interactive Forms are imported.

When importing multiple Interactive Forms, it is highly recommended to check for invalid duplicate field names. This can be done with the function CheckFieldNames(). The function returns the handle of the first field which contains a field name that already exists in use. You can then change the field name with SetFieldNames() and execute CheckFieldNames() again until all invalid field names are changed.

After changing a field name you must also check whether the field is used within a JavaScript Action or function. Such scripts must be changed so that they do not become invalid. Due to the possible references of fields within the script functions and Actions, merging of Interactive Forms is very complicated and should be avoided whenever possible.

**Actions**

Annotations, form fields, bookmarks (also known as outline items), pages, and the global Catalog object may specify actions to perform, such as opening another PDF file, jumping to page, or playing a sound, for example.

Annotations (especially link annotations) and bookmarks can directly be associated with an action. This action will be executed when the object is activated.

Annotations, the catalog object, pages, and form fields support also additional actions which extend the set of events which trigger the execution of an action.

Actions are usually executed in viewer applications only. Otherwise it would be very difficult to understand what happens behind the scenes when editing an object.

Actions can be accessed with the following functions:

- GetObjActions() / GetObjActionsEx()
- GetFirstChildAction() / GetFirstChildActionEx()
- GetJavaScriptActionEx()
- GetMovieAction() / GetMovieActionEx()
- GetNamedAction() / GetNamedActionEx()
- GetCheckAction() / GetCheckActionEx()

GetObjActions() returns the first action of an object and a pointer to the first trigger event if any. Actions and trigger events are stored in a linked list. That means every action and every trigger event can reference another action or event that should be executed.

The actions which must be executed for an object should be copied to an execution list and not directly be executed. This makes sure that a duplicate check can be applied so that no duplicate loop occurs when an action references itself.

The first object that should be examined is the catalog object right after a PDF file was opened with OpenImportFile() or OpenImportBuffer() and after the global objects were imported with ImportCatalogObject().

Now render the first page and add the actions of the page to the event list of this page. After this the actions of form fields should be examined because form fields support events which must be handled when a page was opened, when it become visible or invisible and when it will be closed.

**Digital Signatures**

A digital signature (PDF 1.3) can be used to authenticate the identity of a user and the document's contents. It stores information about the signer and the date and time when it was signed. Once a PDF file was digitally signed, it is impossible to change the file without invalidating the signature. Because of this, it is always possible to check whether a document has been changed or not.

Depending on the Acrobat version certain signatures are handled by Acrobat. DynaPDF supports the PKCS#12 file security handler which is supported since Acrobat 4.0.

**Supported Certificate Formats**

DynaPDF supports internal and external signature handlers. When using the internal signature handler of DynaPDF then you need a PKCS#12 certificate file. Certificates are available in different file formats and different encryption key lengths. DynaPDF supports certificates in the file format PKCS#12 with up to 4096 bit encrypted private/public key pairs on Windows.

On non-Windows operating systems the cross-platform signature library AiCrypto is used to sign PDF files. This signature handler supports 256 bit RSA encrypted private keys only (the AiCrypto library supports almost all available key lengths but it creates undecided length encoded ASN.1 objects for strong encryption key lengths whereas Adobe's Acrobat supports defined length encoded ASN.1 objects only).

The internal signature handler is mainly used with self-sign certificates but it is possible to sign a PDF file with any certificate that is installed on the system's certificate store, including hardware certificates.

**External Signatures**

In order to support software and hardware certificates with almost arbitrary encryption key lengths it is possible to sign a PDF file with an external signature handler. This makes it possible to select a certificate from the system's certificate store and to use system functions, for example, to sign a PDF file.

The function CheckAndSigFileEx(can be used to create detached and non-detached signatures. In case of a non-detached signature CheckAndSigFileEx() returns the SHA1 hash of the PDF file and the external signature handler signs this hash and creates a PKCS#7 signature object that must finally be written to the PDF file with FinishSignature().

A detached signature works almost identically with the exception that the signature handler creates also the hash from the PDF buffer itself. This is recommended for programming languages which support no pointers like C or VB.Net, for example, because an additional copy of the PDF buffer must usually be created and this doubles the memory usage.
PDF/A and PDF/X Compatibility

PDF files can be created for different purposes such as printing, publishing, or archiving which have all their own requirements. Due to these different requirements two separate PDF standards were defined by the ISO Committee, PDF/A and PDF/X.

PDF/X

The PDF/X standard addresses blind exchanges where all files shall be delivered in CMYK (and/or spot colors), with no RGB or device independent color managed data. The ISO common requirement in many areas around the world and in many print sectors – usually tied to an environment where the file supplier seeks to maintain maximum control of the print job. PDF/X 3 is like PDF/X 1 in standard for graphic content exchange. The main difference is that PDF/X 3 allows the use of color management and device-independent color in addition to CMYK and spot colors.

The PDF/X standard requires all fonts to be embedded, the appropriate PDF bounding boxes to be specified, and color to appear as CMYK, spot colors, or both. In addition, PDF/X compliant PDF files must contain information describing the printing condition for which they are prepared (see AddRenderingIntent()).

When creating PDF/X compliant files with DynaPDF you need to know that DynaPDF does not check whether certain features are allowed to use in the selected PDF/X standard. DynaPDF simply writes the required PDF/X key to the file which tells the viewer application that this file is compliant to a specific PDF/X version. Whether this is true or not depends on whether you used all required features and whether all required information were added to the file (e.g. the rendering intent (see AddRenderingIntent()), the document title (see SetDocInfo()) and the trim box for each page (see SetTrimBox()). It is usually best to check the resulting PDF file with a preflight tool before using certain features in a preproduction environment.

However, it is not very difficult to create PDF/X compliant PDF files. The main recommendation is that all form fields are embedded, that at least the trim box for all pages are set, and that colors are defined in the color space DeviceCMYK (see SetColorSpace()). In addition, an ICC profile must be embedded in the file (see AddRenderingIntent()) and images must not be compressed with JPEG2000 compression.

The so-called output PDF version must be set with SetPDFVersion().