DpBio Engine

V1.1.1

DLL Documentation

Table of contents

INTRODUCTION

For Windows programmers

This part of the documentation will give you some general information about how to use the DLL. The only thing you have to include in your projects to use the biometric authentication/identification capabilities of DigiPass BioEngine is “dpBio.dll”. This DLL is compiled with stdcall argument passing convention, it can be used with many programming language, including C, VB, .Net, Delphi, etc.

A header file named “dpBio.h” is also included, it must be used if you are planning to program in C/C++. If you are programming in C/C++ and you want to link “dpBio.dll” statically, you will also need the library file “dpBio.lib. All function descriptions given in this document are for C programming. You should adapt these information to your programming language.

The only file you will have to redistribute with your product is dpBio.dll. This file should be put in the Windows system directory (c:\windows\system32 on windows XP for example), or in your executable directory.

For Linux programmers

This part of the documentation will give you some general information about how to use the static library “libDpBio.a”. The only thing you have to do to use the biometric authentication/identification capabilities of DigiPass BioEngine in your projects is to link them with “libDpBio.a”. This static library is compiled with C argument passing convention, it can be mainly used with C/C++ languages

A header file named “dpBio.h” is also included, it must be used if you are planning to program in C/C++. All function descriptions given in this document are for C programming. You should adapt these information to your programming language.

Since “libDpBio.a” is a static library, you wont have to redistribute any file with you project to use the functions of this SDK.
.

FUNCTIONS DESCRIPTION

NAME	***dpInit***
PROTOTYPE	`int WINAPI dpInit( void );`
DESCRIPTION	Initializes the library (internal memory buffer, pre-compiled data, etc.) Must be called before any other function. If you call a function without a prior call to dpInit the function call will fail.
PARAMETERS	none
RETURN VALUE	ZEF_OK if initialization was ok. An error code listed in the error

	codes table otherwise.

NAME	***dpUnInit***
PROTOTYPE	`int WINAPI dpUnInit( void );`
DESCRIPTION	Free all system resources allocated by the library for its internal processing. Must be called at the end of the calling program. Any dpBio function call made after calling dpUnInit will fail.
PARAMETERS	none
RETURN VALUE	ZEF_OK if uninitialization was ok. An error code listed in the error codes table otherwise.

NAME	***dpGetEngineVersion***
PROTOTYPE	`char*  WINAPI  dpGetEngineVersion( void );`
DESCRIPTION	Returns a null terminating string containing the version number of the bioEngine. The string is a static string, it should not be deallocated.
PARAMETERS	none
RETURN VALUE	A pointer to a null terminating string containing the version number.

NAME	***dpGetMaxTemplateSize***
PROTOTYPE	`int WINAPI dpGetMaxTemplateSize( void );`
DESCRIPTION	Returns the maximum size a template can be. This size can vary if you change the algorithm parameters using loadParameters. So if you have to load custom parameters, always call getMaxTemplateSize after the parameters loading.
PARAMETERS	none
RETURN VALUE	If no error occurred, max size of a template in bytes. An error code listed in the error codes table otherwise.

NAME	***dpGetTemplateSize***
PROTOTYPE	`int WINAPI dpGetTemplateSize( unsigned char *p_templateBuffer );`
DESCRIPTION	Returns the size in bytes of the template passed as argument
PARAMETERS	• p_templateBuffer (IN): A pointer to the buffer containing the template
RETURN VALUE	If no error occurred, size in bytes of the template. An error code listed in the error codes table otherwise.

NAME	***dpGenerateTemplate***
PROTOTYPE	`int WINAPI dpGenerateTemplate( unsigned char *p_imgData, unsigned int p_width, unsigned int p_height, unsigned char *p_templateBuffer );`
DESCRIPTION	Generates a fingerprint template from a fingerprint image. The image must be 8 bit grayscale.
PARAMETERS	• p_imgData (IN): A pointer to the 8 bit greyscale image of a fingerprint. Data must be stored line by line in the buffer with first line being the to of the image. No line padding. • p_width (IN): width of the image in pixels • p_height (IN): height of the image in pixels • p_templateBuffer (OUT): A pointer to the buffer that will receive the fingerprint template. This buffer must be allocated by the caller with enough space to contain one template (see dpGetMaxTempateSize)
RETURN VALUE	ZEF_OK if template generation was ok. An error code listed in the error codes table otherwise.

NAME	***dpVerifyTemplate***
PROTOTYPE	`int WINAPI dpVerifyTemplate( unsigned char *p_templateBuffer1, unsigned char *p_templateBuffer2, unsigned int *p_score );`
DESCRIPTION	Compares two fingerprint templates and indicates if they match or not. Returns also the matching score.
PARAMETERS	• p_templateBuffer1 (IN): A pointer to the first fingerprint template to be compared • p_templateBuffer2 (IN): A pointer to the second fingerprint template to be compared • p_score (OUT): Pointer to the variable that will receive the matching score if the two templates match. Is filled with 0 otherwise.
RETURN VALUE	ZEF_OK if the two template are from the same finger, ZEF_NO_MATCH if they don't match. An error code listed in the error codes table otherwise.

NAME	***dpVerifyAndRefineTemplate***
PROTOTYPE	`int WINAPI dpVerifyAndRefineTemplate( unsigned char *p_templateBuffer1, unsigned char *p_templateBuffer2, unsigned int *p_score, char *p_refined );`
DESCRIPTION	Compares two fingerprint templates and indicates if they match or not. Returns also the matching score. If the matching score is above a certain threshold, the first template will be updated with information coming from the second template. It will generate a composite template containing more information of better quality.
PARAMETERS	• p_templateBuffer1 (IN): A pointer to the first fingerprint template to be compared, will be updated if the match is good • p_templateBuffer2 (IN): A pointer to the second fingerprint template to be compared • p_score (OUT): Pointer to the variable that will receive the matching score if the two templates match. Is filled with 0 otherwise. • p_refined (OUT): Pointer to a boolean that will receive TRUE if the first template has been updated, FALSE otherwise.
RETURN VALUE	ZEF_OK if the two template are from the same finger, ZEF_NO_MATCH if they don't match. An error code listed in the error codes table otherwise.

NAME	***dpGetTemplateQuality***
PROTOTYPE	`int WINAPI dpGetTemplateQuality( unsigned char *p_templateBuffer );`
DESCRIPTION	Return the quality of the fingerprint template passed as argument. Quality ranges from 0 to 100.
PARAMETERS	• p_templateBuffer (IN): A pointer to the fingerprint template whose quality is to be evaluated
RETURN VALUE	If no error occurred, a quality value between 0 and 100. An error code listed in the error codes table otherwise.

NAME	***dpLoadParameters***
PROTOTYPE	`int WINAPI dpLoadParameters( unsigned char *p_buffer, unsigned int p_size );`
DESCRIPTION	Load custom bioEngine parameters. Usually you won't have to use

	this function. But in certain circumstances (for example : to tune the bioEngine to a specific sensor), ZEFYR Technologies could provide you a file to be loaded with this function.
PARAMETERS	• p_buffer (IN): A pointer to the buffer containing the parameters loaded from a file • p_size (IN): size of the parameters data buffer
RETURN VALUE	ZEF_OK if dpLoadParameters executed ok. An error code listed in the error codes table otherwise.

NAME	***dpSetSecurity***
PROTOTYPE	`int WINAPI dpSetSecurity( unsigned int p_securityLevel );`
DESCRIPTION	Set the security level (FAR) to a predefined threshold. If the threshold is very high then security will be very high but there will be more rejection.
PARAMETERS	• p_security level (IN): value of the security level. This value ranges from 0 to MAX_SCORE (default=1000). Typical values are : FAR = 10-0 security level = 0 FAR = 10-1 security level = 25 FAR = 10-2 security level = 50 FAR = 10-3 security level = 75 FAR = 10-4 security level = 100 FAR = 10-5 security level = 125 FAR = 10-6 security level = 150 WARNING: these typical values can change if some custom bioEngine parameters are loaded with dpLoadParameters.
RETURN VALUE	ZEF_OK if dpSetSecurity executed ok. An error code listed in the error codes table otherwise.

NAME	***dpGetSecurity***
PROTOTYPE	`int WINAPI dpGetSecurity( unsigned int *p_securityLevel );`
DESCRIPTION	Retrieves the security level that has been set before by dpGetSecurity
PARAMETERS	• p_security level (OUT): pointer to a variable that will receive the security level. This value will range from 0 to MAX_SCORE (default=1000).
RETURN VALUE	ZEF_OK if dpGetSecurity executed ok. An error code listed in the error codes table otherwise.

NAME	***dpIdentifyTemplate***
PROTOTYPE	`int WINAPI dpIdentifyTemplate( unsigned char **p_templatesArray, unsigned int p_nbTemplates, unsigned char *p_templateBuffer, unsigned int *p_score, unsigned int *p_templatePos );`
DESCRIPTION	Check if there is a match between a fingerprint template and a set of other fingerprint templates. If the algorithm finds a match, it returns the index of the template that match and the matching score.
PARAMETERS	• p_templatesArray (IN): An array containing pointers to buffers where fingerprint templates are stored. This array sis the database in which the candidate template will be searched. • p_nbTemplates (IN): Contains the number of elements in p_templatesArray. • p_templateBuffer (IN): Pointer to a buffer containing the candidate template. • p_score (OUT): will receive the matching score if a match is found. • p_templatePos (OUT): If a match is found, contains the index of the matching template. Otherwise it contains the number of elements in templatesArray (the value of p_nbTemplates) or (in case of error) the index of the template that caused the error.
RETURN VALUE	ZEF_OK if a match was found, if a match was not found the function return ZEF_NO_MATCH. An error code listed in the error codes table otherwise.

NAME	***dpStopIdentification***
PROTOTYPE	`int WINAPI dpStopIdentification( unsigned char *p_templateBuffer );`
DESCRIPTION	This function allows you to stop an identification if you consider it is taking to much time. Its main goal is to allow to cancel identification processes.
PARAMETERS	• p_templateBuffer (IN): Pointer to the buffer that was passed as parameter to dpIdentifyTemplate. This buffer will identify which thread must be ended in case of multiple identification running in different threads.
RETURN VALUE	ZEF_OK if there was no error. An error code listed in the error codes table otherwise.

NAME	***dpSetDbMaxSearch***
PROTOTYPE	`int WINAPI dpSetDbMaxSearch( unsigned int p_maxSearch );`
DESCRIPTION	Set the maximum percentages of the template array database that have to be searched before returning that a match was not found. This function can be useful to speed up a little bit searches with big databases, but you must know that decreasing the search depth can increase the FRR.
PARAMETERS	• p_maxSearch (IN): value ranging from 0 to 100 telling the max percentage of the database to be searched
RETURN VALUE	ZEF_OK if there was no error. An error code listed in the error codes table otherwise.

NAME	***dpSetSpeed***
PROTOTYPE	`int WINAPI dpSetSpeed( unsigned int p_mode, int p_speed );`
DESCRIPTION	Sets the speed of the algorithm. By default the algorithm is set to normal speed which gives the best performance at a reasonable speed. If you really need faster responses, you can accelerate template generation and/or template matching by setting speed to 1 or 2. This will accelerate the algorithm up to 3 times, but will increase the FRR a little.
PARAMETERS	• p_mode (IN): can be one of the following values : - 0 set speed for both template generation and template matching. - 1 set speed for template generation. - 2 set speed for template matching. • p_speed (IN): defined the speed of the algorithm : - 0 normal speed : the algorithm have it's best FAR/FRR performances. - 1 medium speed : the algorithm is faster (up to 1.5X normal speed), but FAR/FRR performances are a little lower. - 2 high speed : the algorithm is really fast (up to 3X normal speed), but FAR/FRR performances are lower.
RETURN VALUE	ZEF_OK if there was no error. An error code listed in the error codes table otherwise.

NAME	***dpProcessPreview***

PROTOTYPE	`int WINAPI dpProcessPreview( unsigned char *p_imgData, unsigned int p_width, unsigned int p_height, unsigned char *p_preview, unsigned int *p_flags );`
DESCRIPTION	Generates a preview image of a fingerprint image. This preview doesn't show the details of the fingerprint and thus preserves the users privacy. This function also generates flags that can be used to help the users fingerprint positioning.
PARAMETERS	• p_imgData (IN): buffer containing the user's fingerprint image. • p_width (IN): witdh of p_imgData. • p_height (IN): height of p_imgData. • p_preview (OUT): buffer that will contain the preview image after the call of the function. Must be allocated withe the same size as p_imgData. • p_flags (OUT): can be a combination of the following flags : - ZEF_FLAG_FINGER_DETECTED : a finger has been detected in imgData. - ZEF_FLAG_REMOVE_FINGER : imgData's quality is good enough. The user can remove his finger from the sensor. - ZEF_FLAG_BEST_IMG : imgData quality is good. This image could be used for fingerprint processing. - ZEF_FLAG_MOVE_UP : The user must move his finger left - ZEF_FLAG_MOVE_DOWN : The user must move his finger down - ZEF_FLAG_MOVE_LEFT : The user must move his finger left - ZEF_FLAG_MOVE_RIGHT : The user must move his finger right - ZEF_FLAG_PRESS_STRONGER : : The user must press his finger stronger on the sensor.
RETURN VALUE	ZEF_OK if there was no error. An error code listed in the error codes table otherwise.

APPENDIX I : ERROR CODES

WARNINGS

NAME	*VALUE*	*DESCRIPTION*
ZEF_NO_MATCH	2	No match was found during authentication/identification
ZEF_TOO_MANY_MINUTIAE	3	Too many minutiae where found during template generation. Some of them where not stored in the template
ZEF_CORE_CLOSE_TO_BORDER	4	A core was found, but too close from the border of the image and so was discarded.

ERRORS

NAME	*VALUE*	*DESCRIPTION*
ZEF_OK	1	Everything worked, no error
ZEF_NULL	0	NULL value
ZEF_NULL_POINTER	-1	A NULL pointer was passed to the function
ZEF_BUFFER_TO_SMALL	-2	The size of the buffer passed as argument is to small
ZEF_OUT_OF_MEMORY	-3	Memory allocation problem. Not enough memory available
ZEF_MATHS_ERROR	-4	Error during math function processing
ZEF_CHECKSUM_FAILED	-5	Fingerprint template checksum verification failed. The template can not be loaded
ZEF_BAD_VERSION	-6	The version of the template that was loaded is incompatible with the current version of the bioEngine. The template can not be loaded.
ZEF_BAD_SECU_THRESHOLD	-7	The security threshold passed as argument was not in the range 0-MAX_SCORE (usually MAX_SCORE =1000)
ZEF_BAD_MAX_SEARCH	-8	The max identification search depth parameter was not in the range 0-100
ZEF_ENGINE_NOT_INITIALIZED	-9	DpInit has not been called
ZEF_BAD_SIZE	-10	The size passed as argument was zero or invalid value
ZEF_NOT_BLOC_MULTIPLE	-11	(Internal error) Image was not cropped to a size multiple of bloc size
ZEF_BAD_SPEED_PARAM	-12	The parameters passed as argument to dpSetSpeed were not valid.
ZEF_STOPPED	-13	The identification process has been stopped by a call to dpStopIdentification

APPENDIX II : ADVANCED ENROLLMENT STATEGIES

Enrollment is a very critical step for fingerprint matching performances. Consequently, special care must be taken for the enrollment procedure. This part of the documentation will give you the recommended enrollment steps for best performances.

The enrollment process consists of four steps, these steps will be described in the following pseudo C code:

```
// 1st step : 1st sample acquisition
// call your image acquisition function... this one is a sample one
myImageAquisitionFunc( imgBuffer, &sizeX, &sizeY );
// generate template
dpGenerateTemplate( imgBuffer, sizeX, sizeY, finalTemplate );

// 2nd step : 2nd sample acquisition
// call your image acquisition function. this one is a sample one
myImageAquisitionFunc( imgBuffer, &sizeX, &sizeY );
// generate template
dpGenerateTemplate( imgBuffer, sizeX, sizeY, workTemplate );
// enhance first template
dpVerifyAndRefineTemplate( finalTemplate, workTemplate, &score, &refined);
// if template was refined, we can go to step 3, otherwise we need
// to see which of the two acquired templates is the best.
// We will keep the best template as finalTemplate
if( !refined )
{
 // get  quality of the two templates
 finalQuality = dpGetTemplateQuality( finalTemplate );
 workQuality = dpGetTemplateQuality( workTemplate );
 // select the template with the highest quality
 if(  workQuality >  finalQuality )
  {
    // copy workTemplate as finalTemplate
    // finaleTemplate MUST have been allocated with the size
    // returned by dpGetMaxTemplateSize.
    // Otherwise you should deallocate  finalTemplate and
    // reallocate it with size = dpGetTemplateSize( workTemplate)
    memcpy( finalTemplate, workTemplate, dpGetTemplateSize( workTemplate) );
  }
}

// 3rd Step : same as 2nd Step
// (this step is not mandatory, but helps getting high quality templates.
...

// 4th Step : verification
// Call your image acquisition function... this one is a sample one
myImageAquisitionFunc( imgBuffer, &sizeX, &sizeY );
```

```
// generate template
dpGenerateTemplate( imgBuffer, sizeX, sizeY, workTemplate );
// verifiy matching between the two templates (and enhance template...)
result = dpVerifyAndRefineTemplate( finalTemplate, workTemplate, &score,
&refined);
// check if verification was successful
if( result == ZEF_OK )
{
 // verification successful,  enrollment is ok !!!
}
else
{
 // verification failed, you should retry the enrollment with
 // the same or another finger.
}
```

APPENDIX III : SAMPLE CODE

Simple console application

This simple sample application source code illustrates the call of all the functions of the SDK. It will be very useful to understand how the functions are used in a real context.

```
#include "stdlib.h"
#include "stdio.h"
#include "dpbio.h"

#define MY_CUSTOM_FILE_ERROR                    -1234
#define MY_SIZE_DONT_MATCH                      -1235
#define MY_COULD_NOT_GET_SIZE_ERROR             -1236
#define MY_COULD_NOT_GET_QUALITY_ERROR          -1237
#define MY_SIZE_DONT_MATCH_ERROR                -1238
#define IMG_WIDTH                               300
#define IMG_HEIGHT                              480



int main( void )
{
```

```
FILE   *file;
int  i,maxTemplateSize,size,fileSize;
int  error,quality;
int  nbTemplates;
unsigned int score,security;
unsigned char *templateArray[2],*image;
char   refined;
unsigned char *params;

// inits
nbTemplates = 10;
params = NULL;
image = NULL;
file = NULL;
for( i=0; i<10; i++ )
  templateArray[i] = NULL;

// initialize bioEngine
error = dpInit();
if( error < ZEF_OK )
  goto errorHandler;

// load some custom parameters (very rare,
// most of the time you will not call this function)

// open file
file = fopen( "customCfg.cfg", "rb" );
if( file == NULL )
 {
  error = MY_CUSTOM_FILE_ERROR;
  goto errorHandler;
 }

// go to the end of the file
error = fseek(file, 0, SEEK_END);
if( error != 0 )
 {
  error = MY_CUSTOM_FILE_ERROR;
  goto errorHandler;
 }
```

```
// get position (=file size)
size = ftell( file );
if( size == -1 )
 {
  error = MY_CUSTOM_FILE_ERROR;
  goto errorHandler;
 }

// allocate memory for config file
params = malloc( size * sizeof( unsigned char) );
if( params == NULL )
 {
  error = ZEF_OUT_OF_MEMORY;
  goto errorHandler;
 }

// read configuration data into buffer
fileSize = fread( params, size, 1, file );
if( size != fileSize )
 {
  error = MY_CUSTOM_FILE_ERROR;
  goto errorHandler;
 }

// close file
fclose(file);
file = NULL;

// release memory
free( params );
params = NULL;

// load some custom parameters
error = dpLoadParameters( params, size );
if( error < ZEF_OK )
  goto errorHandler;

//free params
free( params );
```

```
params = NULL;

// display bioEngine version
printf("Demo application using dpBioEngine version %s\n",
dpGetEngineVersion());

// retrieve max template size for future memory allocations
maxTemplateSize = dpGetMaxTemplateSize();
if( maxTemplateSize < ZEF_OK )
 {
  error = maxTemplateSize;
  goto errorHandler;
 }

// display max template size
printf("\tMax template size is: %d\n", maxTemplateSize);

// allocate memory for some templates
for( i=0; i<nbTemplates; i++ )
 {
  templateArray[i] = malloc(maxTemplateSize*sizeof(unsigned char));
  if( templateArray[i] == NULL )
  {
   error = ZEF_OUT_OF_MEMORY;
   goto errorHandler;
  }
 }

// retrieve old security level
error = dpGetSecurity(&security);
if( error < ZEF_OK )
 {
  error = security;
  goto errorHandler;
 }
printf("\tDefault security level : %d\n",security);

// set security level to 125
error = dpSetSecurity(125);
if( error < ZEF_OK )
  goto errorHandler;
```

```
printf("\tDefault security level set to  125\n");


//allocate memory for image
image = malloc( IMG_WIDTH*IMG_HEIGHT*sizeof(unsigned char) );
if( image == NULL )
 {
  error = ZEF_OUT_OF_MEMORY;
  goto errorHandler;
 }

// load an image... no sample code is given to load images.
// images can be acquired through a sensor or loaded from disk.
// here we will consider that you use a function :
// int getSensorImage( unsigned char *p_buffer, int p_width, int p_height );
// which returns 1 if everything was ok, -1 otherwize

// get an image of size 300x480 in buffer
error = getSensorImage( image, IMG_WIDTH, IMG_HEIGHT );


// generate template from the acquired image
error = dpGenerateTemplate( image, IMG_WIDTH, IMG_HEIGHT, templateArray[0] );
if( error < ZEF_OK )
  goto errorHandler;

// free image memory
free( image );
image = NULL;

// load other template from disk
// open file
file = fopen( "myTemplatePath.bin", "rb" );
if( file == NULL )
 {
  error = MY_CUSTOM_FILE_ERROR;
  goto errorHandler;
 }

// read file, we should check fread error with ferror,
```

```
// but we wont do it for this simple sample app
fileSize = fread( templateArray[1], maxTemplateSize, 1, file );

// close file
fclose( file );
file = NULL;

//get template size from template...
size = dpGetTemplateSize( templateArray[1] );
if( size < ZEF_OK )
 {
  error = size;
  goto errorHandler;
 }

// now check if we have read enough data...
// you should note that if we have not read the full template
// dpGetTemplateSize should have returned with an error
// so this check is a little useless, but lets do it anyway !
if( size != fileSize )
 {
  error = MY_SIZE_DONT_MATCH_ERROR;
  goto errorHandler;
 }

// display templates size and quality
for( i=0; i<2; i++ )
 {
  // get size
  size = dpGetTemplateSize( templateArray[i] );
  if( size < ZEF_OK )
  {
   error = MY_COULD_NOT_GET_SIZE_ERROR;
   goto errorHandler;
  }
  //get quality
  quality = dpGetTemplateQuality( templateArray[i] );
  if( quality < ZEF_OK )
  {
   error = MY_COULD_NOT_GET_QUALITY_ERROR;
```

```
    goto errorHandler;
   }
   printf("\tTemplate %d size = %d",i,size);
   printf("\tTemplate %d quality = %d",i,quality);
 }

 // perform authentication between the two templates
 error = dpVerifyTemplate( templateArray[0], templateArray[1], &score);
 if( error == ZEF_OK )
 {
   printf("\tTemplate 1 matches with template 2 !!! score = %d\n", score);
 }
 else if( error == ZEF_NO_MATCH )
 {
   printf("\tTemplate 1 doesn't match with template 2");
 }
 else
   goto errorHandler;

 // perform authentication with refinement between the two templates
 error = dpVerifyAndRefineTemplate( templateArray[0], templateArray[1], &score,
&refined);
 if( error == ZEF_OK )
 {
   printf("\tTemplate 1 matches with template 2 !!! score = %d\n", score);
   if( refined )
   {
    printf("\tTemplate 1 has be enhanced (refined) with data from template
            2\n");
   }
 }
 else if( error == ZEF_NO_MATCH )
 {
   printf("\tTemplate 1 doesn't match with template 2\n");
 }
 else
   goto errorHandler;

 // lets set the max searching dept for our identification to 100
 // this is already the default value.. we only do this in order to show
 // how to call the function
```

```
 error = dpSetDbMaxSearch( 100 );
 if( error < ZEF_OK )
   goto errorHandler;

 // make an identification in a very little database of 2 templates
 // of course template 1 should match with template 1....
 error = dpIdentifyTemplate( templateArray, &nbTemplates, *
                             templateArray[0], &score );
 if( error == ZEF_OK )
  {
   printf("\tTemplate 1 matches with template %d !!! score = %d\n",
            nbTemplates, score);
  }
 else if( error == ZEF_NO_MATCH )
  {
   printf("\tTemplate 1 doesn't match with any template\n");
  }
 else
   goto errorHandler;

 // uninit bioEngine
 error = dpUnInit();
 if( error < ZEF_OK )
   goto errorHandler;



 // free memory
 for( i=0; i<10; i++ )
   free( templateArray[i] );

 return ZEF_OK;

errorHandler:
// this programing style (handling errors with gotos) can be discussed
// but its quite a useful way of handling errors in an organized way

 // free memory
 for( i=0; i<10; i++ )
  {
   if( templateArray[i] != NULL )
```

```
    free( templateArray[i] );
 }

 if( image != NULL );
 free( image );

 // close file
 if( file != NULL )
   fclose( file );

 // free params
 if( params != NULL )
   free( params );

 // display a message to user...
 printf("Error #%d occurred... exiting !\n",error);

 return error;
}
```

VERSIONS HISTORY

V1.0e (20/01/2005)

- First public version

V1.1.0 (05/10/2005)

- Minor bug fixes
- Speed improvements : template generation and matching are 10% faster
- Added dpSetSpeed function : enable to choose algorithm speed (up to 3x faster than previous algorithm)
- Improved identification algorithm : can be up to 20% faster.
- Improved matching algorithm : FAR/FRR performances are 15% better.
- Added scaling capabilities : The algorithm can now handle images with resolution other than 500 dpi.
- Acquisition helper function dpProcessPreview : build a preview image that can be shown to the user instead of the fingerprint image (to preserve user's privacy) + generates flags that can be used to guide the user during fingerprint image acquisition.
- Fixed a small memory leak in DpGenerateTemplate.

- The BioEngine is now thread-safe. You can run multiple identifications in parallel without getting into troubles.

V1.1.1 (09/03/2006)

- Massive C headers commenting
- Minor bug fixes (no performance improvements)