VeriFinger 4.2 Linux SDK

VeriFinger 4.2 Linux SDK

Table of Contents

1. Introduction 1
2. What's new 4
    2.1. VeriFinger library 4
    2.2. VeriFinger SDK/library 4
3. Fingerprint images 6
4. VeriFinger library 7
    4.1. Library functions 7
    4.2. Error codes 9
    4.3. Registration 11
        4.3.1. VFRegistrationType function 12
        4.3.2. VFGenerateId function 13
        4.3.3. VFRegister function 14
    4.4. Initialization 15
        4.4.1. VFInitialize function 15
        4.4.2. VFFinalize function 16
    4.5. Contexts 16
        4.5.1. VFCreateContext function 18
        4.5.2. VFFreeContext function 19
    4.6. Parameters 19
        4.6.1. VFGetParameter function 24
        4.6.2. VFSetParameter function 25
    4.7. Features extraction 26
        4.7.1. VFExtract function 26
    4.8. Features generalization 28
        4.8.1. VFGeneralize function 29
    4.9. Verification 30
        4.9.1. VFVerify function 30
    4.10. Identification 32
        4.10.1. VFIdentifyStart function 33
        4.10.2. VFIdentifyNext function 34
        4.10.3. VFIdentifyEnd function 34
    4.11. Matching threshold and similarity 35
    4.12. Matching details 35
    4.13. Fingerprint features 38
5. Scanner API 42
6. Sample applications 45
    6.1. Console Demo 45
    6.2. GTK demo 45

Chapter 1. Introduction

VeriFinger 4.2 Linux SDK consists of VeriFinger library, scanner support drivers (API) and sample applications.

- VeriFinger library is a fingerprint recognition engine.
- Scanner drivers provide scanning from fingerprint scanners.
- Sample programs demonstrate SDK features.

VeriFinger library and scanner drivers store fingerprint image in format described in Fingerprint images.

SDK directory contains:

`VFinger/`	The VFinger4.2 library.	
	`VFinger.h`	VeriFinger library header for C/C++ compilers
	`libVFinger.so.4.2.0`	VeriFinger library
	`libVFinger.so.4.2`	VeriFinger library
	`libVFinger.so.4.1.10`	VeriFinger library
	`libVFinger.so.4.1`	VeriFinger library
	`libVFinger.so.4`	VeriFinger library
	`libVFinger.so`	VeriFinger library
`doc/`	VeriFinger library interface description.	

	VeriFinger 4.2 Linux SDK.pdf	This file
samples/	A set of sample images	
scanner/	Scanner drivers	
	Makefile	Make file
	scanner.h	Scanner driver header for C/C++ compilers
	fx2k.c	BiometriKa FX2000 driver implementation (can be used as sample code)
	af-s2.o	AuthenTec AF-S2 driver code
	AFS4000.o	AuthenTec AES4000 driver code
	fx2k.o	BiometriKa FX2000 driver code
fx2k_sdk/	BiometriKa FX2000 Light SDK	
hasp/	The HASP (dongle) drivers	
demo_console/	Subdirectory with simple console demonstrational program	
demo_gtk2/	Subdirectory with sample library demonstrational program	

`LICENSE.txt`	VeriFinger 4.2 SDK license
`README.txt`	ReadMe files for VeriFinger 4.2 Linux SDK
`COPYRIGHT.txt`	VeriFinger 4.2 Linux SDK copyright

Chapter 2. What's new

2.1. VeriFinger library

Version 4.2.0.0

- Improved reliability.
- Better matching performance. Both matching speeds are now about 50% faster than in version 4.1.
- Reduced template size. Now template occupies 150 - 300 bytes (vs. 200 - 650 in version 4.1).
- Features compression and decompression functions are now built in the library.
- Added CrossMatch Verifier 300 scanner mode.
- VeriFinger can now return skeletonized image.

Version 4.1.0.0

- Completely new interface.
- Higher matching speed. Now only two speeds are available - low (0 speed in 4.0) and high (5 speed in 4.0). Both speeds are faster that in version 4.0. For more information see Parameters.
- Improved recognition reliability. Both speeds are more reliable that in version 4.0.
- Optimizations for a number of fingerprint scanners are available. For more information see Parameters.

2.2. VeriFinger SDK/library

Version 4.2.0.2

- Features generalization bug fix
- Minor bugfixes

Version 4.2.0.1

- Minor bugfixes

Version 4.2.0.0

- Uses VeriFinger library version 4.2.0.0.

Version 4.1

- Support for BiometriKa FX2000, AuthenTec AES4000 and AF-S2 scanners

Chapter 3. Fingerprint images

Fingerprint image used by VeriFinger library and scanner drivers has to be an array of bytes of size width*height and pointer to the first element of this array has to be passed to libraries' functions. Lines of the image have to be stored in the array from top to bottom order. Next line must immediately follow the previous one (no padding). Each byte of the array corresponds to fingerprint image pixel (grayscale value). Value of 0 indicates black and value of 255 - white.

Chapter 4. VeriFinger library

VeriFinger library is a fingerprint recognition engine that you can use in your application.

VeriFinger library enables application to implement such scenarios as user enrollment, user verification and user identification using fingerprints. It provides a number of functions to implement such behavior.

When enrolling a user application can use features extraction functions that extracts features from fingerprint image (for more information see Fingerprint images and Features). Also features generalization can be used to increase quality of the features. Then features can be stored in database for later access.

When verifying a user features that are extracted from fingerprint image are compared with etalon features that are in the database or somewhere else. See Verification.

When identifying a user features that are extracted from fingerprint image are compared with all features stored in the database until matching is successful or end of the database passed. See Identification.

VeriFinger library is copy protected. To use it you have to register it. See Registration.

Before using the library it has to be initialized. See Initialization and Contexts.

VeriFinger library behavior is controlled through Parameters.

4.1. Library functions

VeriFinger library contains the following functions grouped by categories:

Registration	
VFRegistrationType	Returns registration type of VeriFinger library
VFGenerateId	Generates registration id from serial number
VFRegister	Registers VeriFinger library
Initialization	

VFInitialize	Initializes VeriFinger library
VFFinalize	Uninitializes VeriFinger library
Contexts	
VFCreateContext	Creates a context
VFFreeContext	Deletes the context
Parameters	
VFGetParameter	Retrieves parameter value
VFSetParameter	Sets parameter value
Features extraction	
VFExtract	Extracts features from fingerprint image
Features generalization	
VFGeneralize	Generalizes count features collections to single features collection
Verification	
VFVerify	Matches two features collections
Identification	
VFIdentifyStart	Starts identification with test features

VFIdentifyNext	Matches with sample features
VFIdentifyEnd	Ends identification

Each of these functions (except for the VFCreateContext) returns integer value to indicate result of the execution. If it is less than zero then execution of the function has failed and the value indicates error code.

Each function (except for registration, initialization and contexts functions) takes last argument of type `HVFCONTEXT`. It is the context in which VeriFinger library functions are called. Pass `null` to use default context. For more information see Initialization and Contexts.

You can use `VFFailed` and `VFSucceeded` functions to determine if the execution of the function failed or succeeded:

C:

```
#define VFFailed(result) ...
#define VFSucceeded(result) ...
```

4.2. Error codes

The following error codes are defined:

General		
`VFE_OK`	0	OK, no error
`VFE_FAILED`	-1	Failed
`VFE_OUT_OF_MEMORY`	-2	Out of memory
`VFE_NOT_INITIALIZED`	-3	VeriFinger library is not initialized
`VFE_ARGUMENT_NULL`	-4	One of the required function arguments is null

VFE_INVALID_ARGUMENT	-5	One of the function arguments is invalid
VFE_NOT_IMPLEMENTED	-9	Function is not implemented
Registration		
VFE_NOT_REGISTERED	-2000	VeriFinger library is not registered
VFE_INVALID_SERIAL_NUMBER	-2001	Specified serial number is invalid
VFE_INVALID_REGISTRATION_KEY	-2002	Specified registration key is invalid
VFE_SCANNER_DRIVER_ERROR	-2003	Scanner driver error
VFE_REGISTRATION_NOT_NEEDED	-2004	No need to register VeriFinger library
VFE_NO_SCANNER	-2005	No scanner found
VFE_MORE_THAN_ONE_SCANNER	-2006	More than one scanner found
VFE_LM_CONNECTION_ERROR	-2007	Error communicating with License Manager
VFE_LM_NO_MORE_LICENCES	-2008	No more License Manager licenses are available
Parameters		
VFE_INVALID_PARAMETER	-10	Parameter identifier is invalid (unknown)
VFE_PARAMETER_READ_ONLY	-11	Parameter is read only
Features extraction		

VFE_ILLEGAL_IMAGE_RESOLUTION	-101	Specified image resolution is illegal
VFE_ILLEGAL_IMAGE_SIZE	-102	Specified image size is illegal
VFE_LOW_QUALITY_IMAGE	-103	Warning. Image quaity is low
VeriFinger specific		
VFE_INVALID_MODE	-1000	Function called in invalid mode
Features		
VFE_INVALID_FEATURES_FORMAT	-3000	Features passed to the function has invalid format

You can use VFErrorToString and VFResultToString functions to get string that describes error and result. These functions are not part of VeriFinger library. They are implemented in VFinger.h and VFingerX.c files.

C:

```
char* VFErrorToString(INT error);
char* VFResultToString(INT result);
void VFCheckResult(INT result);
```

4.3. Registration

You have to register VeriFinger library before using it. If library is not registered all functions (except for initialization, contexts and parameters functions) will return VFE_NOT_REGISTERED. There are several registration types available: not protected library, registration with HASP key, registration to PC, registration to U.are.U scanner and registration in License Manager (LAN protection).

Note
At this moment only registration with HASP key is available in Linux.

If you are using not protected library, you should use it directly without registration.

If you are using registration with HASP key, simply plug it to LPT or USB port before initializing VeriFinger library.

If you are using registration to PC or U.are.U scanner then call VFGenerateId function (for registration with U.are.U scanner connect the scanner before calling the function) and pass serial number provided with your VeriFinger library license to it. This function will generate registration id that you should send to Neurotechnologija (sales@neurotechnologija.com) or distributor from which library was acquired. Then pass serial number with received registration key to VFRegister function.

If you are using LAN protection then you must use string `"LAN"` as serial number and server name as registration key.

To determine how VeriFinger library is registered (and if it needs registration at all) call VFRegistrationType function.

Example:

C:

```
// Registration to PC or to U.are.U scanner
// Your serial number here
CHAR serial_number[] = "xxxx-xxxx-xxxx-xxxx";
// Registration id generation
{
        CHAR registration_id[100];
        VFGenerateId(serial_number, registration_id);
}
// Received registration key
CHAR registration_key[] = "xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx";
// Register VeriFinger library
{
        VFRegister(serial_number, registration_key);
}
```

4.3.1. VFRegistrationType function

Returns VeriFinger library registration type.

C:

```
INT VFINGER_API VFRegistrationType();
```

Return values: Returns one of the following values:

VF_RT_NOT_PROTECTED	0	VeriFinger library is not protected. No need to register
VF_RT_HASP	1	HASP key found either on LPT or USB port. No need to register
VF_RT_PC	2	VeriFinger library is registered to PC
VF_RT_UAREU	4	VeriFinger library is registered to U.are.U scanner
VF_RT_LAN	8	VeriFinger library is registered in License Manager on LAN
VF_RT_UNREGISTERED	6	VeriFinger library is not registered. Call VFRegister function to register

In case of error functions returns VFE_FAILED.

4.3.2. VFGenerateId function

Generates registration id from specified serial number. Serial number and registration id have to be arrays of characters (strings) pointers to first element of each have to be passed to the function. Array for registration id has to be large enough to store the string (100 characters is enough).

C:

```
INT VFINGER_API VFGenerateId(CHAR * serial, CHAR * id);
```

Parameters:

[in]	serial, Serial	Serial number of VeriFinger library license

[out]	id, Id	After execution of the function contains registration id for the serial number

Return values: If serial number or registration id is null returns `VFE_ARGUMENT_NULL`.If serial number is invalid returns `VFE_INVALID_SERIAL_NUMBER`.If serial number indicates registration to DigitalPersona U.are.U scanner then can return `VFE_SCANNER_DRIVER_ERROR` (if there was error communicating with scanner driver), `VFE_NO_SCANNER` (if no scanner detected) or `VFE_MORE_THAN_ONE_SCANNER` (if more than one scanner detected).Otherwise generates registration id and returns `VFE_OK` (in case of error returns `VFE_FAILED`).

4.3.3. VFRegister function

Registers VeriFinger library with specified serial number and registration key. Serial number and registration key have to be arrays of characters (strings) pointers to first element of each have to be passed to the function.

C:

```
INT VFINGER_API VFRegister(CHAR * serial, CHAR * key);
```

Parameters:

[in]	serial, Serial	Serial number of VeriFinger library license. If you have VeriFinger 4.0 PC protection registration key please use customer id instead of serial number. For example: customer id = 11222 then serial will be `"11222"` If you are using LAN protection then you must specify `"LAN"` as serial number.
[in]	key, Key	Registration key for serial number and registration id (received from Neurotechnologija or its distributor). If you have VeriFinger 4.0 PC protection registration key please convert it to string and pass as key. If you are using LAN protection then you must specify server name as registration key.

Return values: If VeriFinger library is not protected or already registered with HASP returns `VFE_REGISTRATION_NOT_NEEDED`.If serial number or registration key is null returns `VFE_ARGUMENT_NULL`.If serial number is invalid returns `VFE_INVALID_SERIAL_NUMBER`.If registration key is invalid (or scanner is not connected for registration to U.are.U scanner) returns `VFE_INVALID_REGISTRATION_KEY`.Otherwise registers VeriFinger library and returns `VFE_OK` (in case of error returns `VFE_FAILED`).

4.4. Initialization

VeriFinger library requires initialization to be performed before any function call and uninitialization to be performed after all function calls (except for contexts functions). This is performed using VFInitialize and VFFinalize functions.

Each successful call to `VFInitialize` should have a corresponding call to `VFFinalize`. So you can call `VFInitialize` more than one time, but you have to call `VFFinalize` equal number of times.

Also you may not call initialization functions at all if you will not work with default context, only with your custom context.

See Contexts for more information.

Example:

C:

```
// Main application function
{
        // Application initialization code
        VFInitialize();
        // Other application code
        VFFinalize();
        // Application uninitialization code
}
```

4.4.1. VFInitialize function

Creates default context by calling VFCreateContext function and initialized VeriFinger library.

C:

```
INT VFINGER_API VFInitialize();
```

Return values: If succeeded return value indicates number of times function have been called before. If it first call to the function return value will be zero.If default context was not created returns `VFE_OUT_OF_MEMORY`.

4.4.2. VFFinalize function

Destroys default context by calling VFFreeContext function and uninitializes VeriFinger library if call to the function corresponds to first call to VFInitialize function.

C:

```
INT VFINGER_API VFFinalize();
```

Return values: Return value indicates number of times function should be more called (number of `VFInitialize` calls without `VFFinalize` calls).If VeriFinger library was not initialized returns `VFE_NOT_INITIALIZED`

4.5. Contexts

Context is a set of parameters and internal structures that VeriFinger library functions use. They are created with VFCreateContext function and destroyed with VFFreeContext function.

Contexts enable different application parts to work with VeriFinger library simultaneously. Inside one context no VeriFinger functions should be called simultaneously because they are not guaranteed to be thread-safe. VeriFinger functions called in different context are guaranteed to be thread-safe.

Parameters are set for the context. So you can use contexts not only to ensure that your application is thread safe, but also use different parameters in different situations. For example you can perform features extraction for different scanners in different contexts with custom set of parameters. For more information see Parameters.

Also, several VeriFinger library functions should be called in specific order. If you have started identification (VFIdentifyStart) then you cannot call functions that work with internal matching structures (except for the VFIdentifyNext) such as VFSetParameter, VFGeneralize, VFVerify and VFIdentifyStart in the same context until you call VFIdentifyEnd. And you cannot call `VFIdentifyNext` and `VFIdentifyEnd` before you call `VFIdentifyStart` in the same context. In these situations functions will return `VFE_INVALID_MODE`.

Example: Working from different threads:

C:


```
// First thread function
{
        // Create context
        HVFCONTEXT context = VFCreateContext();
        // Call VeriFinger library functions, for example
        VFVerify(..., context);
        // Delete context
        VFFreeContext(context);
}
// Second thread function
{
        // Create context
        HVFCONTEXT context = VFCreateContext();
        // Call VeriFinger library functions, for example
        VFIdentifyNext(..., context);
        // Delete context
        VFFreeContext(context);
}
```

Example: Contexts with different parameters:

C:

```
HVFCONTEXT context1; // First context
HVFCONTEXT context2; // Second context
// Initialization function
{
        // Set parameters for default context
        VFSetParameter(..., NULL);
        VFSetParameter(..., NULL);
        // Create first context
        context1 = VFCreateContext();
        // Set parameters for first context
        VFSetParameter(..., context1);
        VFSetParameter(..., context1);
        // Create second context
        context2 = VFCreateContext();
        // Set parameters for second context
        VFSetParameter(..., context2);
        VFSetParameter(..., context2);
}
// Some application function
{
        HVFCONTEXT context;
        if (/* image from first scanner */) context = context1;
```



```
        else
                if (/* image from second scanner */)
                        context = context2;
                else
                        context = NULL; // default context
        // Call VeriFinger library functions, for example
        VFExtract(..., context);
}
// Uninitialization function
{
        // Delete first context
        VFFreeContext(context1);
        // Delete second context
        VFFreeContext(context2);
}
```

Example: Wrong functions call order:

C:

```
// Some application function
{
        //...
        VFIdentifyStart(...);
        for (...)
        VFIdentifyNext(...);
        VFVerify(...); // Error, returns VFE_INVALID_MODE
        VFIdentifyEnd(...);
        //...
        VFExtract(...);
        VFIdentifyNext(...); // Error, returns VFE_INVALID_MODE
}
```

4.5.1. VFCreateContext function

Creates context with default parameters.

C:

```
HVFCONTEXT VFINGER_API VFCreateContext();
```

Return values: Return value is newly created context.If context cannot be created returns

VFE_OUT_OF_MEMORY.

4.5.2. VFFreeContext function

Deletes context created with VFCreateContext.

C:

```
INT VFINGER_API VFFreeContext(HVFCONTEXT context);
```

Parameters:

	context, Context	Context to delete

Return values: If context is null returns VFE_ARGUMENT_NULL else returns VFE_OK.

4.6. Parameters

Some VeriFinger algorithm aspects are controlled through parameters. Parameters are retrieved and set for the specified context by VFGetParameter and VFSetParameter functions. Some parameters are read only (informational). If you will try to set a read only parameter VFSetParameter function will return VFE_PARAMETER_READ_ONLY. If you will pass an invalid parameter identifier to one of these functions it will return VFE_INVALID_PARAMETER.

Parameters can be of the following types:

Referenced as	Size (bytes)	VF_TYPE_XXX constant	C equivalent
Void		VF_TYPE_VOID0	
Byte	1	VF_TYPE_BYTE1	BYTE
Signed byte	1	VF_TYPE_SBYTE2	SBYTE
Word	2	VF_TYPE_WORD3	WORD

Short integer	2	VF_TYPE_SHORT4	SHORT
Double word	4	VF_TYPE_DWORD5	DWORD
Integer	4	VF_TYPE_INT6	INT
Boolean	4	VF_TYPE_BOOL10	BOOL
Char	1	VF_TYPE_CHAR20	CHAR
String	4	VF_TYPE_STRING100	CHAR*

To determine parameter type call VFGetParameter function with parameter identifier VFP_TYPE and value - needed parameter identifier. Also you may use VFGetParameter-Type function. Result of the function will be one of VF_TYPE_XXX constants.

When retrieving a parameter value pass pointer to variable of parameter type as value for VFGet-Parameter function.

For string parameter pass pointer to first char in the string as value. To retrieve length of the string (not including the terminating null character) pass null as value. Function will return length of the string.

When setting a parameter value pass the value casted to double word to VFSetParameter function.

The following parameter identifiers are defined (grouped by categories):

Identifier	Value	Read only	Type	Description
General				
VFP_TYPE	0	x		See parameters types earlier

VFP_NAME	10	x	String	Name of the VeriFinger library
VFP_VERSION_HIGH	11	x	Double word	Major version of VeriFinger library
VFP_VERSION_LOW	12	x	Double word	Minor version of VeriFinger library
VFP_COPYRIGHT	13	x	String	Copyright of VeriFinger library
Features extraction				
VFP_EXTRACT_FEATURES	110		Integer	Obsolete, will be removed in the next version
VFP_RETURNED_IMAGE	10002		Integer	Specifies what image features extraction function will return. Can be one of the following:
VF_RETURNED_IMAGE_NONE	0	None image is returned - the image will be the same as passed to features extraction function		
VF_RETURNED_IMAGE_BINARIZED	100	Binarized image will be returned (default)		
VF_RETURNED_IMAGE_SKELETONIZED	200	Skeletonized image will be returned		
Features matching (Verification and Identification)				

VFP_MATCHING_THRESHOLD	200		Integer	Minimal similarity of two features collections that are identical. Must be not less that zero. See also Matching threshold
VFP_MAXIMAL_ROTATION	201		Integer	Maximal rotation of two features collection to each other. Must be in range VFDIR_0..VFDIR_180. See also information about directions in Features and Matching details
VFP_MATCH_FEATURES	210		Integer	Obsolete, will be removed in the next version
VFP_MATCHING_SPEED	220		Integer	Speed of features matching. Can be one of the following:
VF_MATCHING_SPEED_LOW	0	Low matching speed		
VF_MATCHING_SPEED_HIGH	256	High matching speed		
Features generalization				
VFP_GENERALIZATION_THRESHOLD	300		Integer	Has the same meaning for features generalizatiom as VFP_MATCHING_THRESHOLD parameter for features matching. See also Matching threshold

VFP_GEN_MAXIMAL_ROTATION	201		Integer	Has the same meaning for features generalization as VF_MAXIMAL_ROTATION parameter for features matching
VeriFinger specific				
VFP_MODE	1000		Integer	Specifies mode in which VeriFinger algorithm is operating (optimized parameter set) Can be one of the following:
VF_MODE_GENERAL	0	General		
VF_MODE_DIGITALPERSONA_UAREU	100	DigitalPersona U.are.U		
VF_MODE_BIOMETRIKA_FX2000	200	BiometriKa FX2000		
VF_MODE_KEYTRONIC_SECUREDESKTOP	300	Keytronic SecureDesktop		
VF_MODE_IDENTIX_TOUCHVIEW	400	Identix TouchView		
VF_MODE_PRECISEBIOMETRICS_100CS	500	PreciseBiometrics 100CS		
VF_MODE_STMICROELECTRONICS_TOUCHCHIP	600	STMicroelectronics TouchChip		
VF_MODE_IDENTICATORTECHNOLOGY_DF90	700	IdenticatorTechnology DF90		

VF_MODE_AUTHENTEC_AFS2	800	Authentec AFS2
VF_MODE_AUTHENTEC_AES4000	810	Authentec AES4000
VF_MODE_ATMEL_FINGERCHIP	900	Atmel FingerChip
VF_MODE_BMF_BLP100	1000	BMF BLP100
VF_MODE_SECUGEN_HAMSTER	1100	SecuGen Hamster
VF_MODE_ETHENTICA	1200	Ethentica
VF_MODE_CROSSMATCH_VERIFIER300	1300	CrossMatch Verifier 300

4.6.1. VFGetParameter function

Retrieves specified parameter value for specified context.

C:

```
INT VFINGER_API VFGetParameter(INT parameter, VOID * value, HVFCONTEXT
context);
```

Parameters:

	parameter, Parameter	Parameter identifier to retrieve
[out]	value, Value	Pointer to variable that will receive parameter value.
	Context, Context	Context to retrieve parameter from. Null for default context

Return values: If context is null and VeriFinger library is not initialized returns VFE_NOT_INITIALIZED.If parameter is invalid (unknown) returns VFE_INVALID_PARAMETER.If value is null returns VFE_ARGUMENT_NULL. For string parameters returns length of the string (not including the terminating null character).Otherwise returns VFE_OK.

Example:

C:

```
// Some application function
{
        CHAR *name;
        INT len;
        DWORD version;
        INT vfp_mode_type;
        INT mode;

        // Get VeriFinger library name
        len = VFGetParameter(VFP_NAME, NULL, NULL);
        name = (CHAR*)malloc((len + 1) * sizeof(CHAR));
        VFGetParameter(VFP_NAME, name, NULL);
        printf(name);
        free(name);

        // Get VeriFinger library major version
        VFGetParameter(VFP_VERSION_HIGH, version, NULL);
        printf("Version: %u.%u", HIWORD(version), LOWORD(version));

        // Determine parameter VFP_MODE type
        vfp_mode_type = VFGetParameter(VFP_TYPE, (VOID*)VFP_MODE, NULL);

        // returned value: VF_TYPE_INT
        // Get integer parameter VFP_MODE value
        VFGetParameter(VFP_MODE, &mode, NULL);
        printf("Mode: %d", mode);
}
```

4.6.2. VFSetParameter function

Sets specified parameter value for specified context.

C:

```
INT VFINGER_API VFSetParameter(INT parameter, DWORD value, HVFCONTEXT
context);
```

Parameters:

	parameter, Parameter	Parameter identifier to set
	value, Value	Parameter value to set
	context, Context	Context to set parameter to. Null for default context

Return values: If context is null and VeriFinger library is not initialized returns `VFE_NOT_INITIALIZED`.If identification is started returns `VFE_INVALID_MODE`.If parameter is invalid (unknown) returns `VFE_INVALID_PARAMETER`.Otherwise returns `VFE_OK`.

Example:

C:

```
// Some application function
{
        // Set VFP_MODE parameter to VF_MODE_DIGITALPERSONA_URU
        VFSetParameter(VFP_MODE, (DWORD)VF_MODE_DIGITALPERSONA_URU, NULL);
}
```

4.7. Features extraction

You can use features extraction to extract features from fingerprint image and then store them in a database (enroll fingerprint). For more information see Fingerprint images and Features.

Use VFExtract function to perform features extraction.

4.7.1. VFExtract function

Extracts features from fingerprint image in the specified context.

Image has to be an array of bytes of size width*height and pointer to the first element of this array has to be passed to this function. Image resolution has to be passed to the function. Function resizes image to resolution of VF_IMAGE_RESOLUTION dpi (dots per inch) internally. Filtered image that is returned by the function is resized back to original resolution. Features returned by the function have resolution of VF_IMAGE_RESOLUTION dpi, so if you wish to display features on the image you have to draw features on the image resized to VF_IMAGE_RESOLUTION dpi.

Features have to be an array of bytes and pointer to the first element of the array has to be passed to this function. Number of bytes occupied by features in the array will be returned by the function in size (Size) parameter. If array size is less than needed then behavior of the function is undefined. To ensure that array is large enough set its size to at least VF_MAX_FEATURES_SIZE. For more information see Features.

The function uses features extraction and VeriFinger specific Parameters.

C:

```
#define VF_IMAGE_RESOLUTION500
#define VF_MIN_IMAGE_RESOLUTION50
#define VF_MAX_IMAGE_RESOLUTION5000
#define VF_MIN_IMAGE_DIMENSION16
#define VF_MAX_IMAGE_DIMENSION2048
#define VF_MAX_FEATURES_SIZE 10000
INT VFINGER_API VFExtract(INT width, INT height, BYTE * image, INT resolution,
BYTE * features, DWORD * size, HVFCONTEXT context);
```

Parameters:

	width, Width	Fingerprint image width. After resize have to be in range from VF_MIN_IMAGE_DIMENSION to VF_MAX_IMAGE_DIMENSION
	height, Height	Fingerprint image height. After resize have to be in range from VF_MIN_IMAGE_DIMENSION to VF_MAX_IMAGE_DIMENSION
[in/out]	image, Image	Fingerprint image to extract features from. After execution of the function contains filtered image
	resolution, Res-	Resolution of the fingerprint image (in dots per inch). Must

	olution	be in range from VF_MIN_IMAGE_RESOLUTION to VF_MAX_IMAGE_RESOLUTION
[out]	features, Features	After execution of the function contains features extracted from fingerprint image
[out]	size, Size	After execution of the function contains size of the features in bytes.
	context, Context	Context to perform features extraction in. Null for default context.

Return values: If VeriFinger library is not registered returns VFE_NOT_REGISTERED.If context is null and VeriFinger library is not initialized returns VFE_NOT_INITIALIZED.If fingerprint image width or height is not in legal range returns VFE_ILLEGAL_IMAGE_SIZE.If resolution is not in legal range returns VFE_ILLEGAL_IMAGE_RESOLUTION.If image, features or size is null returns VFE_ARGUMENT_NULL.Otherwise performs features extraction and returns either VFE_OK or VFE_LOW_QUALITY_IMAGE (if fingerprint image quality is low). VFE_LOW_QUALITY_IMAGE is only a warning. Calling application can either ignore it or ask the user to rescan the fingerprint.

Example:

C:

```
// Extraction function
{
        INT width, height;
        BYTE *image;
        INT resolution;
        BYTE features[VF_MAX_FEATURES_SIZE];
        DWORD size;
        // Load the image from file or get the image from scanner
        // ...
        VFExtract(width, height, image, resolution, features, &size, NULL);
}
```

4.8. Features generalization

You can use features generalization to increase quality of the recognition. Generalization performs conjunction of several features collections to one collection, validates each feature and removes noisy features. You can use features generalization during enrollment. To obtain features for generalization use Features extraction functions.

Generalization uses `VF_GENERALIZATION_THRESHOLD` parameter for matching to determine if provided features collections are of the same finger. For more information see Matching threshold.

Use VFGeneralize function to perform features generalization.

4.8.1. VFGeneralize function

Performs generalization of features collections in the specified context. Currently generalization can be performed only for `VF_GENERALIZE_COUNT` features collections.

This function uses features extraction, features generalization, features matching and VeriFinger specific Parameters.

C:

```
#define VF_GENERALIZE_COUNT 3
INT VFINGER_API VFGeneralize(INT count, const BYTE * const * gen_features,
BYTE * features, DWORD * size, HVFCONTEXT context);
```

Parameters:

	count, Count	Count of features collections to generalize
[in]	gen_features, GenFeatures	Array of features collections to generalize
[out]	features, Features	After execution of the function contains generalized features
[out]	size, Size	After execution of the function contains size of generalized features in bytes.
	context, Context	Context to perform features generalization in. Null for default context.

Return values: If VeriFinger library is not registered returns VFE_NOT_REGISTERED.If context is null and VeriFinger library is not initialized returns VFE_NOT_INITIALIZED.If identification is started returns VFE_INVALID_MODE.If count of features collections is other that VF_GENERALIZE_COUNT returns VFE_INVALID_ARGUMENT.If features collections are null returns VFE_ARGUMENT_NULL.If one of the passed features collections has invalid format returns VFE_INVALID_FEATURES_FORMAT.If features collections cannot be generalized returns VFE_FAILED.Otherwise return index of features collection on which base features generalization has been performed.

Example:

C:

```
// Generalization function
{
        BYTE *feats[3];
        BYTE features[VF_MAX_FEATURES_SIZE];
        DWORD size;
        feats[0] = /*obtain first fingerprint features*/;
        feats[1] = /*obtain second fingerprint features*/;
        feats[2] = /*obtain third fingerprint features*/;
        if (VFSucceeded(VFGeneralize(3, feats, features, &size, NULL))
                printf("Generalization succeeded");
        else
                printf("Generalization failed");
}
```

4.9. Verification

You can use verification to determine if two features collections are of the same finger. It uses VFP_MATCHING_THRESHOLD parameter (See Matching threshold). To obtain features from fingerprint image use Features extraction functions. Also you may use features generalization functions to increase recognition reliability.

Use VFVerify function to perform verification.

4.9.1. VFVerify function

Performs two features collections verification in the specified context.

Pass pointer to matching details structure that will receive details of features collections matching (set size (Size) member before calling the function to actual size of the structure). Pass null if you are not interested in matching details. For more information see Matching details.

This function uses features matching and VeriFinger specific parameters. For more information see Parameters.

C:

```
INT VFINGER_API VFVerify(const BYTE * features1, const BYTE * features2,
VFMATCHDETAILS * md, HVFCONTEXT context);
```

Parameters:

[in]	features1, Features1	First fingerprint features
[in]	features2, Features2	Second fingerprint features
[in/out]	md, MD	After execution of the function contains details of features collections matching
	context, Context	Context to perform verification in. Null for default context.

Return values: If VeriFinger library is not registered returns `VFE_NOT_REGISTERED`.If context is null and VeriFinger library is not initialized returns `VFE_NOT_INITIALIZED`.If identification is started returns `VFE_INVALID_MODE`.If one of the features collections is null returns `VFE_ARGUMENT_NULL`.If one of the passed features collections has invalid format returns `VFE_INVALID_FEATURES_FORMAT`.If insufficient memory then returns `VFE_OUT_OF_MEMORY`.If features collections similarity is high enough (see `VFP_MATCHING_THRESHOLD` in Parameters) returns `VFE_OK` (the same finger features collections). Otherwise returns `VFE_FAILED`.

Example:

C:

```
// Verification function
{
        BYTE *features1, *features2;
        VFMATCHDETAILS md;
        BOOL result;
```

```
        features1 = /*obtain first fingerprint features*/;
        features2 = /*obtain second fingerprint features*/;
        md.size = sizeof(md);
        result = VFSucceeded(VFVerify(features1, features2, &md, NULL));
        if (result)
                printf("Same finger. Similarity: %d", md.similarity);
        else
                printf("Different fingers. Similarity: %d", md.similarity);
}
```

4.10. Identification

Use identification to identify fingerprint in the database.

First start identification with unknown fingerprint (test) features. Use VFIdentifyStart function.

Then walk through all database features (sample features) and match them with test features (with VFIdentifyNext function) until matched (function returns `VFE_OK`) or end of the database passed. It uses `VFP_MATCHING_THRESHOLD` parameter (see Matching threshold).

You may also use G to increase speed of the identification: match first sample features which G is equal to test features G; then sample features which G difference with test features is 1, then with G difference 2 and so on. It is a quite high probability that fingerprint will be identified during first matches if it is in the database. See also Features.

End the identification (VFIdentifyEnd function).

To obtain features for identification use Features extraction function (for enrollment in the database you may also use features generalization functions).

Example:

C:

```
// Identification function
{
        BYTE *test_features;
        BYTE *sample_features;
        BOOL found;
        test_features = /*obtain features of fingerprint to identify*/;
        VFIdentifyStart(test_features, NULL);
        found = FALSE;
        for (/* walk through database */)
```

```
        {
                sample_features = /*some features from the database*/;
                if (VFSucceeded(VFIdentifyNext(sample_features, NULL, NULL)))
                {
                        found = TRUE;
                        break;
                }
        }
        VFIdentifyEnd(NULL);
        if (found)
                printf("Fingerprint found in the database");
        else
                printf("Fingerprint not found in the database");
}
```

4.10.1. VFIdentifyStart function

Starts identification with specified test features in specified context.

This function uses features matching and VeriFinger specific parameters. For more information see Parameters.

C:

```
INT VFINGER_API VFIdentifyStart(const BYTE * test_features, HVFCONTEXT
context);
```

Parameters:

[in]	test_features, Test-Features	Test features
	context, Context	Context to start identification in Null for default context.

Return values: If VeriFinger library is not registered returns `VFE_NOT_REGISTERED`.If context is null and VeriFinger library is not initialized returns `VFE_NOT_INITIALIZED`.If identification is started returns `VFE_INVALID_MODE`.If test features collection has invalid format returns `VFE_INVALID_FEATURES_FORMAT`.If test features are null returns `VFE_ARGUMENT_NULL`.If insufficient memory then returns `VFE_OUT_OF_MEMORY`.

4.10.2. VFIdentifyNext function

Matches sample features with test features in specified context.

Pass pointer to matching details structure that will receive details of features collections matching (set size (Size) member before calling the function to actual size of the structure). Pass null if you are not interested in matching details. For more information see Matching details.

This function uses features matching and VeriFinger specific parameters. For more information see Parameters.

C:

```
INT VFINGER_API VFIdentifyNext(const BYTE * sample_features, VFMATCHDETAILS *
md, HVFCONTEXT context);
```

Parameters:

[in]	sample_features, SampleFeatures	Sample features
[in/out]	md, MD	After execution of the function contains details of features collections matching.
	Context, Context	Context to perform features matching in. Null for default context.

Return values: If VeriFinger library is not registered returns `VFE_NOT_REGISTERED`.If context is null and VeriFinger library is not initialized returns `VFE_NOT_INITIALIZED`.If identification is not started returns `VFE_INVALID_MODE`.If sample features are null returns `VFE_ARGUMENT_NULL`.If test features collection has invalid format returns `VFE_INVALID_FEATURES_FORMAT`.If features collections similarity is high enough (see `VFP_MATCHING_THRESHOLD` in Parameters) returns `VFE_OK` (the same finger features collections). Otherwise returns `VFE_FAILED`.

4.10.3. VFIdentifyEnd function

Ends the identification started with VFIdentifyStart function in specified context.

C:


```
INT VFINGER_API VFIdentifyEnd(HVFCONTEXT context);
```

Parameters:

	context, Context	Context to end identification in. Null for default context.

Return values: If VeriFinger library is not registered returns `VFE_NOT_REGISTERED`.If context is null and VeriFinger library is not initialized returns `VFE_NOT_INITIALIZED`.If identification is not started returns `VFE_INVALID_MODE`.

4.11. Matching threshold and similarity

VeriFinger features matching algorithm provides value of features collections similarity as a result. It can be obtained in matching details. The higher is similarity, the higher is probability that features collections are obtained from the same finger fingerprints.

You can set matching threshold - the minimum similarity value that verification and identification functions accept for the same finger fingerprints. You can set the matching threshold using `VFP_MATCHING_THRESHOLD` parameter (`VFP_GENERALIZATION_THRESHOLD` for features generalization).

Matching threshold is linked to false acceptance rate (FAR, different fingers fingerprints erroneously accepted as the of the same finger) of matching algorithm. The higher is threshold, the lower is FAR and higher FRR (False Rjection Rate, same finger fingerprints erroneously accepted as different fingers fingerprints) and vice a versa. You can use `VFMatchingThresholdToFAR` and `VFFARToMatchingThreshold` functions to convert, matching threshold to FAR in percents and vice a versa. Only values of and for FAR between 1% and 0.001% can be calculated more or less correctly. All other values are calculated very approximately. These functions are not part of VeriFinger library; for C they are implemented in `VFingerX.h` and `VFingerX.c` files.

C:

```
double VFMatchingThresholdToFAR(INT th);
INT VFFARToMatchingThreshold(double f);
```

4.12. Matching details

Matching details describes relationship between pair of feature collections determined during verification or identification. Matching details are defined as structure, pointer to which you can pass to verification or identification functions. It can be one of the following structures. When passing to the function set size (Size) member to size of actual structure and cast pointer to the structure to pointer to the first structure. See also Features.

C:

```
typedef struct _VFMatchDetails
{
        DWORD Size;
        INT Similarity;
        INT Rotation;
        INT TranslatonX;
        INT TranslationT;
} VFMatchDetails;

#define VF_MAX_MATCHED_MINUTIA_COUNT 1024
typedef struct _VFMatchedMinutiae
{
        INT Count;
        SHORT Test[VF_MAX_MATCHED_MINUTIA_COUNT];
        SHORT Sample[VF_MAX_MATCHED_MINUTIA_COUNT];
} VFMatchedMinutiae;

typedef struct _VFMatchDetailsEx
{
        DWORD Size;
        INT Similarity;
        INT Rotation;
        INT TranslationX;
        INT TranslationY;
        VFMatchedMinutiae MM;
} VFMatchDetailsEx;
```

Members:

Size	Size of the structure. Set this member before calling a function
Similarity	Similarity between pair of feature collections. The bigger is the value the higher is similarity. See also Matching threshold and similarity

Rotation	Rotation of two features collections to each other. It is an angle in range [0, VFDIR_360). See also information about directions in Features
TranslationX	Translation between two features collections along X axis
TranslationY	Translation between two features collections along Y axis
MM.Count	Count of minutiae common for two features collections in MM collection
MM.Test MM.Sample	Matched minutiae arrays (common minutiae for two features collections). First - index of minutia in test (first) features collection, second - index of minutia in sample (second) features collection. See also Features

Example:

C:

```
//Identification function
{
        BYTE * testFeatures;
        BYTE * sampleFeatures;
        VFMatchDetails md;

        //...
        VFIdentifyStart(testFeatures, NULL);
        md.Size = sizeof(md);
        for (...)
        {
                VFIdentifyNext(sampleFeatures, &md, NULL);
                printf("Similarity: %d", md.Similarity);
        }
        VFIdentifyEnd(NULL);
}

        // Verification function
{
        BYTE * features1, * features2;
        VFMatchDetailsEx md;
```

```
        INT I;

        // ...
        md.Size = sizeof(md);
        VFVerify(features1, features2, (VFMatchDetails *)&md, NULL);
        for (i = 0; i < md.MM.Count; i++)
        printf("Matched minutia %d: index in first features - %d; index in
second features - %d\n", i, md.MM.Test[i], md.MM.Sample[i]);
}
```

4.13. Fingerprint features

Features or features collection or template are data extracted from fingerprint image that is used in verification and identification. To obtain features from fingerprint image use Features extraction functions. You may also use features generalization to improve quality of the features.

Features are stored in array of bytes. Size of the array will never exceed `VF_MAX_FEATURES_SIZE`.

You can use `VFFeatGetXxx` and `VFFeatSet` functions to decompress and compress features.

C:

```
typedef enum _VFSingularPointType
{
        vfsptUnknown = 0,
        vfsptCore = 1,
        vfsptDoubleCore = 2,
        vfsptDelta = 3
} VFSingularPointType;

typedef struct _VFSingularPoint
{
        INT X;
        INT Y;
        VFSingularPointType T;
        BYTE D;
} VFSingularPoint;

typedef enum _VFMinutiaType
{
        vfmtUnknown = 0,
        vfmtEnd = 1,
```

```
        vfmtBifurcation = 2
} VFMinutiaType;

typedef struct _VFMinutia
{
        INT X;
        INT Y;
        VFMinutiaType T;
        BYTE D;
        BYTE C;
        BYTE G;
} VFMinutia;

// Features compression
INT VFINGER_API VFFeatSet(BYTE g, INT mCount, const VFMinutia * m,
INT spCount, const VFSingularPoint * sp, INT boWidth,
INT boHeight, const BYTE * bo, BYTE * features);

// Features decompression
INT VFINGER_API VFFeatGetG(const BYTE * features);
INT VFINGER_API VFFeatGetMinutiaCount(const BYTE * features);
INT VFINGER_API VFFeatGetMinutiae(const BYTE * features, VFMinutia * m);
INT VFINGER_API VFFeatGetSPCount(const BYTE * features);
INT VFINGER_API VFFeatGetSP(const BYTE * features, VFSingularPoint * sp);
INT VFINGER_API VFFeatGetBOSize(const BYTE * features, INT * pWidth, INT *
pHeight);
INT VFINGER_API VFFeatGetBO(const BYTE * features, BYTE * bo);
```

All functions take features (Features) argument as compressed features and/or m (M) argument as minutia array (mCount (MCount) is length of the array), sp (SP) argument as singular point array (spCount (SPCount) is length of the array), bo (BO) argument as blocked orientation image (in similar format as fingerprint image; boWidth (BOWidth) and boHeight (BOHeight) are accordingly width and height of blocked orientations image). `VFFeatSet` function returns size of compressed features.

Features consist of:

- G - obtained using `VFFeatGetG` function
- Minutiae - obtained using `VFFeatGetMinutiae` function (number of minutiae obtained using `VFGetMinutiaCount` function)
- Singular points - obsolete, obtained using `VFFeatGetSP` function (number of singular points obtained using `VFGetSPCount` function). VeriFinger does not extract singular

points

- Blocked orientations - obsolete, obtained using `VFFeatGetBO` function (size of blocked orientations obtained using `VFGetBOSize` function). VeriFinger does not extract blocked orientations.

G: G is a global fingerprint feature that reflects ridge density. It can have values from 0 to 255, so it occupies one byte. The bigger is value the bigger is ridge density. You can use `VFFeatG` function to get ridge density.

Minutiae: Minutiae are points in fingerprint image where finger ridges end or separate. Each minutia is a structure with the following members: X, Y - coordinates in pixels, T - type, D - direction, C - curvature, G - g.

Singular points: Singular points are locations in fingerprints image where finger ridges screw. Each singular point is a structure with the following members: *X*, *Y* - coordinates in pixels, *T* - type, *D* - direction.Each minutia and singular point has x and y coordinates (from top-left corner of the image) and direction.Direction is byte value in range [`VFDIR_0`, `VFDIR_360`). To convert it to degrees multiply by 180 and divide by `VFDIR_180` and vice a versa to convert degrees to direction. Also you may use `VFDirToDeg` and `VFDegToDir` functions. To convert them to radians and vice a versa use `VFDirToRad` and `VFRadToDir` functions. Following constants are defined:

VFDIR_0	0	0°
VFDIR_45	30	45°
VFDIR_90	VFDIR_45 * 2	90°
VFDIR_135	VFDIR_45 * 3	135°
VFDIR_180	VFDIR_45 * 4	180°
VFDIR_225	VFDIR_45 * 5	225°
VFDIR_270	VFDIR_45 * 6	270°
VFDIR_315	VFDIR_45 * 7	315°

VFDIR_360	VFDIR_45 * 8	360°
VFDIR_UNKNOWN	127	Unknown direction
VFDIR_BACKGROUND	255	Background

Blocked orientations: Blocked orientations are ridges orientation of every fingerprint image block of `VF_BLOCK_SIZE` * `VF_BLOCK_SIZE` pixels. Can be byte value range [`VFDIR_0`, `VFDIR_180`), `VFDIR_UNKNOWN` or `VFDIR_BACKGROUND`.

C:

```
// Conversion from VFDIR_XXX to degrees and vice a versa
#define VFDirToDeg(dir) ...
#define VFDirToDegF(dir) ...
#define VFDegToDir(deg) ...

// Conversion from VFDIR_XXX to radians and vice a versa
#define VFDirToRad(dir) ...
#define VFRadToDir(a) ...

// Orientation stuff
#define VFIsBadArea(orient) ...
#define VFIsGoodArea(orient) ...
#define VFTheOrient(orient) ...
#define VFIsUnknown(orient) ...
#define VFIsOrient(orient) ...
```


Chapter 5. Scanner API

Scanner API is very simple and can be easily understood from `scanner.h` file.

This file has `scanner_info` structure:

```
struct scanner_info
{
        int dpi;
        char *name;
        int version;
        void (*close) (void);
        int (*init) (void);
        unsigned char * (*read) (int *width, int *height);
};
extern struct scanner_info scanner_AFS4000;
extern struct scanner_info scanner_af_s2;
extern struct scanner_info scanner_fx2k;
```

Currently VerFinger Linux SDK has support for AFS4000, AF-S2, and FX 200 scanners.

Every scanner driver has exported name pointing to scanner_info structure. This structure is the same for every scanner.

scanner_info structure description:

Structure member	Description
int *dpi*	Specifies scanners resolution in DPI.
char **name*	Scanner name in human readable format (null-terminated string).
int *version*	Driver version.
void (**close*) (void)	Function used to close harwdare device after use.
int (**init*) (void)	Function used to open hardware device before

	use. Returns 0 if operation was successful.
unsigned char * (*read) (int *width, int *height)	Image scanning. Return NULL, if no image can be received from scanner or returns pointer to image if scan was successfull. width and height can't be NULL, returns image size. Returned memory can't be released, and is overwritten with second scan.

Important

These drivers are not thread safe.

Example:

```
#include <stdlib.h>
#include <scanner.h>
struct scanner_info* scanner = &scanner_AFS4000;
int main (void)
{
        int w, h;
        char *image;

        // open device
        if (scanner->init())
        return 1; // initialization error

        // scan image
        image = scanner->read(&w, &h);
        if (image == NULL)
                return 2; // scannning error

        // close device
        scanner->close();

        // return OK
        return 0;
}
```

This sample can be compiled using GCC compiler:

```
gcc sample.c -Iscanner scanner/AFS400.o -o sample -lusb
```

`AFS400.o` requires `libusb` and must be linked with `-lusb`

`afs-s2.o` requires libusb and must be linked with `-lusb`

`fx2k.o` must be linked with `-lfx3L -lfx3scan -lfxoem -lm`

Chapter 6. Sample applications

VeriFinger Linux SDK contains two samples: simple console demo and GUI sample based on GTK library.

Console demo program demonstrates simple VeriFinger usage. This is very good start point.

Gtk demo demonstrates usage of VeriFinger library and scanner drivers. It allows to test enrollment and identification using scanner or image files.

6.1. Console Demo

This demo shows how to use VeriFinger library. It loads TIFF files, extracts fingerprint features performs comparison. This demo uses VFExtract function to extract features from fingerprint image and compares these images using function VFVerify. For identification demo utilizes VFIdentifyStart, VFIdentifyNext and VFIdentifyEnd functions.

This demo requires `libtiff` to load TIFF files.

To compile demo you need `libtiff` header files.

Compile:

```
make
```

Run:

```
make run
```

If VeriFinger library is installed on system, please type:

```
./vf_console_demo
```

or set LD_LIBRARY_PATH

```
export LD_LIBRARY_PATH="../VFinger:${LD_LIBRARY_PATH}
```

and run vf_console_demo

6.2. GTK demo

This demo shows how to use VeriFinger library and scanners. This demo can input data from image file or fingerprint scanner. This demo also shows how to change VeriFinger algorithm parameters.

This demo shows how to:

- Use VFGetParameter and VFSetParameter functions
- Decompress and visualize fingerprint features.
- Extract features using VFExtract.
- Perform identification (VFIdentifyStart, VFIdentifyNext and VFIdentifyEnd).
- Work with low quality images.
- Generalize features using VFGeneralize function
- Use scanner API.

This demo supports background scanning and can be used for presentations. It suports simple data base, which can be loaded from file or saved back.

Demo requires GTK+ 2.0 at minimum, and GTK+ 2.2.4 is recomended. As some problems were detected using GTK+ 2.2.2 we do not recommend use of this version (sometimes GTK doesn't close dialog windows).

AF-S2 and AES4000 scanner driver require `libusb` library. These drivers and demo has been tested on 2.4.x and 2.6.x Linux kernels.

To run this demo:

GTK+ 2 and VeriFinger libraries must be installed.

VeriFinger library can be installed into system or exported environment variable:

```
LD_LIBRARY_PATH="../VFinger:${LD_LIBRARY_PATH}" ./vf_gtk_demo
```

or type

```
make run
```


To compile this demo:

`GTK+ 2` and `GTK+-2-dev` libraries must be installed on system. `libusb-dev` library must be installed if fingerprint scanner will be used. To compile run:

```
make
```