AuthenTec, Inc.
Personal Security for the Real World™

Programmer's Reference Manual

Minutiae
047 018 287
106 021 192
070 023 210
053 024 000
073 032 230
108 032 428
091 033 174
058 039 248
108 054 402
125 059 400
099 060 400
070 061 256
048 068 340
065 070 338
104 071 358
115 075 358
041 077 096
123 079 384
063 083 064
053 091 052
028 097 052
084 100 031
050 102 044
103 106 050
117 111 046
104 118 282

…for Microsoft® Windows

2060 Rev 1.5
15 Aug 00

AuthenTec, Inc.
Post Office Box 2719
Melbourne, Florida 32902-2719
321-308-1300
www.authentec.com

AuthenTec welcomes your input. We try to make our publications useful, interesting, and informative, and we hope you will take the time to help us improve them. Please send any comments or suggestions by mail or e-mail.

AuthenTec, Inc.
Post Office Box 2719
Melbourne, Florida 32902-2719
321-308-1300
www.authentec.com
apps@authentec.com

Programmer's Reference Manual for Microsoft Windows
2060 Rev 1.5 (15AUG00)

Table of Contents

INTRODUCTION 5

FINGERPRINT BIOMETRIC PROGRAMMING OVERVIEW 5
TEMPLATE STORAGE OVERVIEW 8
FAS Internal Template Database 9
API GROUP OVERVIEW 10
SOURCE CODE EXAMPLES 12

THE HIGH PERFORMANCE API 13

THEORY OF OPERATION 14
API FUNCTION CALLS 15
Event Messages 16
Synchronous Operation Implementation 17
Asynchronous Operation Implementation 17
APPLICATION EXAMPLE 19

THE FINGERPRINT SERVICES API 20

THEORY OF OPERATION 21
API FUNCTION CALLS 22
Event Messages 23
Synchronous Operation Implementation 25
Asynchronous Operation Implementation 26
APPLICATION EXAMPLE 27

THE CONVENIENCE API 28

THEORY OF OPERATION 29
API FUNCTION CALLS 29

THE IMAGE CAPTURE API 30

API FUNCTION CALLS 30

THE UTILITY AND COMMON API 32

API FUNCTION CALLS 32

EVENT MESSAGES, ERROR CODES, AND RETURN CODES 34

EVENT MESSAGES 34
ERROR CODES 34
RETURN CODES 35

THE HIGH PERFORMANCE API DEMONSTRATION 36

DESCRIPTION 36
File 37
Enroll User 37
Validate User ID 40
Verify Template 41
Identify User 41
Help 41

IMPLEMENTATION 42

API REFERENCE 47

FingerLocAllocOSBitmap 48
FingerLocClose 49
FingerLocCloseStream 50
FingerLocConvertRawImageToOSBmp 51
FingerLocDeleteOSBitmap 52
FingerLocDeleteUser 53
FingerLocEnroll 54
FingerLocGetCurrentImage 56
FingerLocGetImageState 57
FingerLocGetLeds 58
FingerLocGetNextUser 59
FingerLocGetResultDetails 60
FingerLocGetVersion 61
FingerLocIdentify 62
FingerLocInit 63
FingerLocOpenStream 64
FingerLocReleaseImage 65
FingerLocResample150 66
FingerLocResample200 67
FingerLocSetLeds 68
FingerLocValidateFingers 69
FingerLocValidateID 70
FLAbortTransaction 71
FLBeginAcquireImage 72
FLBeginEnroll 73
FLBeginIdentify 75
FLBeginValidateID 76
FLBeginVerify 77
FLBuildMatchTemplate 79
FLBuildOSBmpHeader 80
FLBuildReferenceTemplate 81
FLContinueTransaction 82
FLConvertNormalReferenceTemplateToSmall 83
FLEndAcquireImage 84
FLEndEnroll 85
FLEndIdentify 86
FLEndIdentifyEx 87
FLEndValidateID 88
FLEndValidateIDEx 89
FLEndVerify 90
FLEndVerifyEx 91
FLGetCommPort 92
FLGetImageBufferSize 93
FLGetMatchTemplateSize 94
FLGetOSBmpHeaderSize 95
FLGetReferenceTemplateSize 96
FLGetSmallReferenceTemplateSize 97
FLMatchTemplatePair 98
FLMatchTemplatePairEx 99
FLReadCommPort 101
FLReleaseCommPort 102
FLReleaseMessage 103
FLWriteCommPort 103

Introduction

The Application Program Interface (API) software detailed in this document includes extensive functional features and services, enabling evaluators and customers to integrate the advanced TruePrint™ Technology of the FingerLoc™ Authentication System (FAS) directly into their application programs.

The API allows other programs running on the same personal computer (or network) to communicate with the features and services of the FingerLoc sensor IC. It allows fingerprint identification processes to be called from any application that can call a Microsoft® Windows® DLL, such as C and C++ personal computer programs.
As a matter of policy, AuthenTec continuously evaluates the various proposed standards for biometric system APIs (such as the Human Authentication - Application Programming Interface (HA-API) of the U.S. Department of Defense). Naturally, this research has a pervasive influence on the general course of our product development.

In this spirit, we actively solicit the opinions and advice of our evaluators and customers regarding this and other API releases, especially regarding standards and requirements for expanded API services.

Fingerprint Biometric Programming Overview

This document makes reference to many terms and operations that are used within the fingerprint biometrics field. This section provides an overview of these terms and operations, not only from a programmer's perspective, but also how they apply to the FingerLoc Authentication System.

The FAS is a combination of the sensor IC required to scan a finger to obtain a fingerprint, and the underlying software components necessary for sensor control, image data collection, fingerprint template creation, fingerprint template storage, and fingerprint template matching and identification. The Dynamic Optimization™ features designed into TruePrint Technology enables fingers with a wide range of physical characteristics such as wetness, dryness, age, and skin type to be successfully scanned for their respective fingerprint. In addition, this same technology can obtain fingerprint scans on fingers coated with contaminants such as oils, dirt, dust, and other contaminants.

The FAS supplies three API groups in order to accomplish these operations: High Performance API, Fingerprint Services API and the Convenience API. Additionally, there are two other APIs supplied to assist an application in other tasks as well. These five API groups, listed below, are discussed in detail in the sections to follow.

- High Performance
- Fingerprint Services
- Convenience
- Image Capture
- Utility and Common

Application programs integrate fingerprint biometric technology by exercising specific types of operations, or primary processes, that are grouped into four categories:

- **Enrollment** This is the process by which a fingerprint template is created or extracted from the scanned image of a fingerprint and subsequently stored in an internal or external database. The stored template is designated as the "reference template" and is compared against fingerprint templates (called "match templates") that are acquired by the FAS during future fingerprint authentication operations (that is, Identification, Verification, and Validation).

  The fingerprint template contains the necessary metrics required to mathematically describe the respective fingerprint. Each of the High Performance API, Fingerprint Services API, and Convenience API provides a mechanism to enroll fingers. Refer to the respective sections of this manual for details on how to implement the enrollment process for each API. Refer to the "Template Storage Overview" section within this manual for more information regarding FAS templates.

- **Identification** This is the process by which a finger placed on the sensor is scanned, and the resulting image processed, to produce a fingerprint template (match template). The template is then compared to the reference templates stored in a database. A mathematical correlation between the match template and the reference template is used to identify the person (user) that has placed their finger on the sensor ("Who am I?).

  Each of the High Performance API, Fingerprint Services API, and Convenience API provides one or more or methods to accomplish identification operations. The High Performance API and the Convenience API provide designated *Identification* functions that use the internal FAS database for template comparison. The High Performance API and the Fingerprint services API can also perform

identification using external databases. Refer to the respective sections of this manual for details on how to implement the identification process for each API. Identifying a single user from a large fingerprint database can be a very time-consuming operation.

The FAS operates with two types of fingerprint templates: a larger template that enables high-speed identification, and a smaller template for use where external storage is limited and look-up speed is not the primary issue. Use of these templates is described in the Template Storage Overview section of this manual. Refer to the respective High Performance API, the Convenience API, and the Fingerprint Services API sections of this manual to get information on the actual size of the small and large exported templates.

- **Verification** — In this process, a finger placed on the sensor is scanned and the resulting image processed to produce a fingerprint template (a "match" template). The match template is then compared to an external reference template that has been passed in from the calling application. A mathematical correlation between the match template and the reference template verifies that the person with the finger on the sensor corresponds to the person described by the reference template ("Am I who I say I am?"). This approach is very useful in situations where the user is known and it is necessary to verify that the person claiming to be the user is not an imposter.

  Each of the High Performance API, the Fingerprint Services API, and the Convenience API provides one or more methods to accomplish *Verification* operations. Refer to the respective sections of this manual for details on how to implement this process for each API.

Validation — In this process, a reference template for a designated user is obtained from a database. A finger is then placed on the sensor and scanned to produce a temporary fingerprint template (match template). The match template is then compared against the reference template. A mathematical correlation between the match template and reference template validates that the person with their finger on the FingerLoc sensor IC corresponds to the person described by the reference template ("Am I the person in the data base?").

Validation is very similar to *Verification*, the difference being that in *Validation* a reference template is already stored in an internal database. Rather than passing an entire template, a simple User ID is passed and subsequently the corresponding template is looked up from an internal database. The High Performance API and the Convenience API both provide functions to accomplish a *Validation* operation.

Template Storage Overview

The FAS creates fingerprint templates from images acquired by scanning a finger with the FingerLoc sensor IC. A fingerprint template contains only the necessary metrics required to mathematically describe its parent fingerprint. Fingerprint templates stored in a database for future use are called “reference templates”. Temporary fingerprint templates that are generated by the FAS during normal use are called “match templates”. Match templates are compared to reference templates during the various identification, verification and validation operations and destroyed when the operation is complete. Reference templates are created either by calling the *Enrollment* function of the High Performance API or the Convenience API, or by calling the *build reference template* procedure of the Fingerprint Services API.

Fingerprint templates can be stored either internally or externally. Internal templates are automatically managed by the FAS and stored in a special database. Internal templates are added to the database by the FAS during an *Enrollment* and removed from the database by the FAS in response to a *Delete User* request from an application program. Templates that are stored externally must be stored and managed by the application program. Optionally, external templates are exported to the application program from the *Enrollment* functions of the High Performance API and Convenience API, or by the *Build* functions of the Fingerprint Services API.

It is the responsibility of the application program to present the externally stored template to the FAS whenever the application wishes to execute an identification, verification, or validation operation. Additionally, some application solutions may use a storage scheme that combines combination internal and external template storage. All templates, whether stored internally or externally, are compressed and encrypted. The format of an actual fingerprint template is proprietary. Refer to the respective sections of this manual that describe the High Performance API, the Convenience API, and the Fingerprint Services API for more information on creating, storing, and exporting templates using the *Enrollment* and *Build* functions.

The FAS uses two different sizes of fingerprint templates, classed as “large: and “small”. Large templates provide additional information that enables high-speed (one-to-many) user identification from large template databases. Small templates are used in solutions where memory space requirements are limited and high-speed identification is not crucial (such as “smart card” applications). The FAS uses a large size as its native mode when creating or internally storing templates. All API functions that perform *Enrollment* or that build match or reference templates from fingerprint images only create large templates. Thus all internally stored templates and exported (external) templates created are of the large size. Applications that wish to store small fingerprint templates must convert the template exported by an *Enrollment* function (from the High Performance API or Convenience API) or a template created using a *Build* function (Fingerprint Services API) to a small template using the proper conversion functions prior to storage. Any *Identification*, *Verification*, or *Validation* operation that uses reference templates stored in the internal database will automatically use large templates.

Conversely, an application program can perform an *Identification* or a *Verification* operation (*Validation* not valid for externally stored templates) by presenting either large or small templates to the FAS. The FAS uses high-speed look-up of the User ID when an application presents a list of externally stored fingerprint templates that consists of only large templates during an *Identify* operation. High-speed User ID look-up is not used if the application program presents a list of externally stored fingerprint templates that consists of a mixture of small and large templates, or all small templates, during an identify operation. Refer to the respective

High Performance API, the Convenience API, and the Fingerprint Services API sections of this manual to get information on the actual size of the small and large exported templates.

FAS Internal Template Database

As previously stated, internal templates are added to the database by the FAS during an *Enrollment* process, and removed from the database by the FAS in response to a *Delete User* request from the application program. The *Enrollment* functions found in the High Performance API and the Convenience API can be called with parameters that specify to save the fingerprint template resulting from the *Enrollment* operation into the internal FAS template database.

Fingerprint templates for a specific finger are stored as a sub-record for the User to whom the finger belongs. Users are assigned a User ID by the application program when a call is made to enroll a finger. The User ID assigned to each user must be unique, otherwise the *Enrollment* function will return an error. The FAS refers to internally stored user information by the unique User ID. The User ID is embedded as a member of the **tsFL_ID_INFO** structure as shown below.

```
typedef struct
{
        uint16  iIDStructSize ; // field contents not required
        int8    iUserId[24] ;   // Null terminated 23 char User ID for this user
        int8    iDisplayName[74] ;      // Display Name for this user (optional)
} tsFL_ID_INFO;
```

The User ID can be any sequence of bytes with a length not longer than 23, and must be terminated with a NULL. Optionally, an application can assign a sequence of NULL-terminated bytes, up to 74 (including the NULL terminator) characters, to the **iDisplayName** member of the **tsFL_ID_INFO** structure. The information in this field is simply stored in the User's record for later use by the application program. As indicated by the nomenclature of the actual structure member, various applications use this field to store the User's actual ASCII name (first and last) in order to associate the User to the User ID. The **iDisplayName** field cannot be modified once the user is enrolled for the first time.

In addition, the *Enrollment* functions require a finger number to be supplied as an input parameter. The finger number passed in can be any **int16** value. It is the responsibility of the application program to provide it's own finger number scheme to associate the fingerprint template being stored to the physical finger.

The application creates a new User in the FAS internal template database by enrolling the User and the finger with a User ID that does not currently exist (the finger number must also be specified). The application can add a new finger to an existing User by calling the desired API *Enrollment* function using the exact User ID of the existing User record, then specifying a finger number that does not currently exist for the existing user. Existing fingerprint templates for a specific finger number can be replaced by re-enrolling a finger and specifying the exact User ID for the existing user and the exact finger number of the existing fingerprint template to be replaced.

The application can request that a specific User be removed from the internal FAS database by calling the **FingerLocDeleteUser** function (of the Utility and Common API) and supplying the desired User ID. All fingerprint templates currently stored for the deleted User are also

deleted. There is no method to delete an individual fingerprint template currently stored for an existing User without deleting the entire User record.

An application program can use the **FingerLocGetNextUser** function (Utilities and Common API) to build a list of all Users and their associated User ID and Display Names from the User records that are stored in the FAS internal database. A list is built by calling this function in a loop until an end-of-Users condition is signaled. This looping method can be used to obtain a count of the total number of Users stored in the internal FAS database. Currently no other information regarding the Users and/or their associated fingerprint templates can be retrieved. Design changes are in-progress to provide applications with a much more robust set of database management features.

API Group Overview

The FAS features and services provide five different groups of API functions. These are intended to address the needs of various API users by enabling access to the sensor IC and biometric services in many different ways.

The scope of these function groups range from the Image Capture API, which presumes that the application has a non-AuthenTec fingerprint-matching algorithm, to the Convenience API in which the FAS features and services do virtually all of the work. All of the API functions are available all of the time, but the functions are meant to work in concert with the others in their group.

The following API service groups and specific API services are provided in this release:

- **The High Performance API**
  This set of functions allows a user interface application to take full advantage of the streamed images and real-time adaptability of the FAS to provide optimum ability to acquire difficult fingerprints and exceptional ability to adapt to changing finger conditions and environments (Dynamic Optimization).

  Some advantages of this API group over the Fingerprint Services API, described below, include simplified operation (integrated sensor and matcher system is easier to use) and optional use of internal or external template storage databases. This group provides the highest performance possible while still giving the application complete control over the Graphical User Interface (GUI).

- **The Fingerprint Services API**
  This set of functions provides a user interface application, or calling application, with the ability to perform similar operations to legacy APIs typically used with optical sensor technology. In addition, the interface also maximizes the use of the Dynamic Optimization features available in the sensor IC during scanning operations. This interface is useful to speed migration from existing legacy systems or in special cases where the design of the interface fits best for a specific solution.

  Unlike the High Performance API, the various operations of the Fingerprint Services API do not provide simplified operations of an integrated sensor and matcher system. The Fingerprint Services API does however provide an advantage in networked systems where an acquired match image can be collected by a remote sensor and

shipped over the network to a central processor responsible for reference template storage and matching.

- **The Convenience API**
  This set of functions provides an application with a simplistic one-step mechanism to perform primary operations such as *Enrollment*, *Identification*, *Verification*, and so on. The function calls of this API provide their own GUI interface objects, such as feedback windows and interactive dialog box controls.

  The Convenience API is designed to provide a calling application with a set of functions that maximize the use of the TruePrint Technology Dynamic Optimization features available in the sensor IC during finger scanning operations. The Convenience API allows a means to quickly construct demonstrations directly from applications in order to evaluate the sensor. Most developers will want to replace functionality provided by this group of API functions with that of either the High Performance or Fingerprint Services APIs for their final product.

- **The Image Capture API**
  This set of functions allows the calling application to receive images only. It includes the functions necessary to open the sensor stream, close the sensor stream, get the current image status, and get the current image. No Dynamic Optimization is performed when using the functions of this API.

- **The Utility and Common API**
  The Utility and Common API provide the programmer with several commonly needed utility functions. Functions for image re-sampling, image format conversion, database manipulation, program initialization, sensor communication control and error reporting are included.

All of the APIs work on a client/server basis. The FAS loads supporting software when the operating system is booting, and the API interfaces to that data object. This client/server arrangement allows multiple applications to access the system. (FingerLoc features and services work with only one application at a time, but applications are pushed and popped into the server's context.)

The following sections describe each of the API groups in detail, with function prototypes and diagrams where appropriate.

Source Code Examples

Several sample programs are included with the Software Developer's Kit (SDK.) These programs were installed in the same directory selected for the SDK. The sample code is in the **\FingerLoc** sub-directory under **\samples.**

If your application uses Microsoft Foundation Classes (MFC) in a shared DLL, and your application is in a natural language other than the current language of the operating system, you must copy the corresponding localized resources **MFC42xxx.DLL** from the Microsoft® *Visual C++* compact disc to…

```
\WINDOWS\SYSTEM\MFCLOC.DLL
```

or to…

```
\WINDOWS\SYSTEM32\MFCLOC.DLL
```

In which the **xxx** in the resource file name represents the three-letter language abbreviation. (for example, **MFC42DEU.DLL** contains resources translated into the German (**Deu**tsch) language). If you don't do this, some of your application's user interface elements will remain in the language of the operating system.

The High Performance API

The High Performance API is designed to provide a user interface application, or calling application, with a suite of functions that will maximize use of the TruePrint Technology Dynamic Optimization features available in the sensor IC during finger scanning operations. In addition, the High Performance API provides start-to-finish automated control of the primary processes, such as *Enrollment*, *Verification*, and *Identification*, once the process has been initiated by the calling application.

The High Performance API provides the following functionality:

- **Enrollment** (creation of reference templates) of User's fingers.
- **Identification** to look up which, if any, user stored in the internal, or external, database belongs to a finger on the sensor.
- **Validation** that the finger on the sensor belongs to a specified user according to the specified user's template stored in the internal database.
- **Verification** that a finger on the sensor matches that of a user finger template stored in an external template or list of external templates passed in.

The automation features of the High Performance API makes it unnecessary for the calling application to manipulate any type of low level sensor objects such as collecting raw fingerprint images, or creating reference or match templates. Fingerprint templates created by the High Performance enrollment process can be stored in the internal database, exported to the calling application for external storage or both stored internally and exported. *Identification*, *Validation,* and *Verification* operations can be called once a finger has been enrolled.

The application enrolls Users simply by calling the *Enrollment* function of the API and passing it the desired User and finger ID information in addition to specifying whether the created fingerprint template should also be stored in the internal FAS database (templates are always made available for export). The calling application is signaled when the operation is complete and the operation's results are available.

Identification, *Validation*, and *Verification* are accomplished in a similar manor. The application simply calls the desired API function while passing the necessary information and the FAS automatically executes the entire operation. The calling application is signaled when the operation is complete and that the results are available.

These design features and abilities simplify the usage of the FAS by the calling application. The High Performance API should be used for any authentication solutions that are located on a standalone system where the sensor IC and the database that stores the reference templates for the users and their fingerprints reside on the same device.

Theory of Operation

The High Performance API uses a transaction-based methodology to interact with the calling application. Each API operation is initiated by the application program with a *Begin* type function call and subsequently terminated with an *End* type function call; for example, **FLBeginEnroll** and **FLEndEnroll**. Once an operation (transaction) has begun, event messages are sent to the calling application from the FAS. These event messages provide the calling application with feedback information as to the progress of the current transaction. The event messages contain various items such as fingerprint images that can be used for display, finger placement information, sequencing information and so on. The calling application processes these event messages in order to update user displays and to control various stages of the current transaction. The FAS sends a *Data Ready* event message to indicate that the operation is complete. The calling application processes the *Data Ready* event message and in turn calls the transaction's corresponding *End* function to obtain the final results and end the current transaction.

The calling application can receive the event messages from the FAS using either a synchronous or asynchronous method. The synchronous method requires the application to call into the FAS to obtain each event message. Once obtained, the application processes the event message and calls a subsequent *Release* function to release the message. This procedure is then repeated for the next event message. The asynchronous method requires the calling application to pass a pointer to a callback function as a parameter to the transaction's *Begin* function call. The FAS will in turn call the *Callback* function with event messages at the time the event message is produced (asynchronous to the calling application's process execution). The application's *Callback* function provides the code necessary to process the various event messages. A call to *Release* the message must also be included within the *Callback* procedure.

There is an advantage to using the asynchronous event message processing method over the synchronous event message processing method. Asynchronous event message processing allows the calling application to initiate a transaction and continue with its normal path of execution. The main application remains free for processing other tasks during the open transaction. However, it is not always possible to correctly implement the required *Callback* function necessary to enable the asynchronous method. The synchronous method requires the thread of the calling application to loop while polling for event messages during the open transaction. This method makes it more difficult for the application program to process other tasks, such as user input, in between event message processing.

An application can prematurely terminate an in-progress transaction by calling the High Performance API *Cancel Transaction* function.

The basic operational flow for the two modes is described in the following sections.

> **Note:** Most of the functions in the High Performance API can time out. When this occurs, the server sends a message indicating that a timeout has occurred.

API Function Calls

The functions in the High Performance API group are shown below.

- **FLBeginEnroll** - This function initiates an *Enrollment* operation - that is, create a fingerprint template for - a finger for a specified user. Several parameters are passed into this function including the User ID, the number of the finger to be enrolled, and so on.
- **FLEndEnroll** – This function retrieves the results from an *Enrollment* operation that is currently in progress, once the results are available. The exported template is also made available to the calling application, in addition to the results of the *Enrollment*.
- **FLBeginValidateID** – This function initiates a *Validation* operation to determine if the finger placed on the FingerLoc sensor IC matches a finger that belongs to a User whose User ID in the internal database corresponds to that which is passed into the function
- **FLEndValidateID** – This function retrieves results from a *Validation* operation that is currently in progress, once the results are available.
- **FLEndValidateIDEx** – This function retrieves results from a *Validation* operation that is currently in progress, once the results are available. In addition, the function returns the strength of the match results, based on FAR/FRR.
- **FLBeginVerify** – This function initiates a *Verification* operation to verify that the finger placed on the FingerLoc sensor IC matches one of the fingerprint templates passed into the function from an external template or list of templates.
- **FLEndVerify** – This function retrieves the results from a *Verification* operation that is currently in progress, once the results are available.
- **FLEndVerifyEx** – This function retrieves the results from a *Verification* operation that is currently in progress, once the results are available. In addition, the function returns the strength of the match, based on FAR/FRR.
- **FLBeginIdentify** – This function initiates an operation to look up or identify the User ID, if any, of the User whose finger matches that of the finger that is placed on the sensor. This *Identification* operation uses templates stored in the internal database for the comparison process.
- **FLEndIdentify** – This function retrieves the results from an *Identification* operation that is currently in progress, once the results are available.
- **FLEndIdentifyEx** – This function retrieves the results from an *Identification* operation that is currently in progress once the results are available. In addition the function returns the quality scoring values that were used to make the match.

- **FLContinueTransaction** – This function retrieves a pointer to the next event message sent from the FAS when an event is available while using the High Performance API in the synchronous mode of operation. Function returns a NULL if an event message is not available at the time of the call. Once a valid message is processed the **FLReleaseMessage** function must be called in order to delete the message.

- **FLReleaseMessage** – This function releases (deletes) the memory space allocated by a valid message that was returned by a previous call to **FLContinueTransaction** while operating in (synchronous mode) or a message that was received by a callback function while operating in asynchronous mode.

- **FLAbortTransaction** – This function aborts the current operation (transaction) and returns the FAS to an idle state. Calling this function will abort any operation (transaction) currently in progress. This function requires the transaction ID that was returned by the operation originally initiated with a *Begin* function called.

The functions in this group are listed alphabetically and described in detail in the "API Reference" section of this manual.

Event Messages

Event messages are passed throughout the system using a pointer to a **tsFL_API_MSG** event message structure as shown below. The first member of the event message structure contains the message type. The second parameter contains the sub-message number for the respective message type. Only a few message types actually have a sub-message number. The third parameter contains a pointer to the actual message data.

This data depends upon the message type. For example, if the message type is an **FL_API_NEW_DISPLAY_IMAGE**, the **pMessageData** pointer points to a buffer that contains a raw fingerprint image that can be formatted and displayed by the calling application. Details of the message types, sub-message numbers, and message data information can be found in the header file **FLStdPI.h**. The table below outlines the various message types that can be expected when processing event messages.

```
typedef struct
{
        uint16  uiMessageType ;
        uint16  uiSubMessageNum ;
        void*   pMessageData ;
} tsFL_API_MSG;
```

MESSAGE TYPE	DESCRIPTION
FL_API_NEW_DISPLAY_IMAGE	Sensor has a new image to be displayed.
FL_API_CLEAR_DISPLAY_IMAGE	Clear the last displayed image.
FL_API_NEW_STATE_TEXT	The state of the sensor has changed - display a new text message.
FL_API_CLEAR_STATE_TEXT	Sensor state text message is no longer valid.
FL_API_PLAY_PROMPT_SOUND	Play sound for action completion, end of capture, capture rejection, and so on.
FL_API_NEW_PROMPT_TEXT	Prompt the user for a finger action.
FL_API_CLEAR_PROMPT_TEXT	Remove any user prompts from the display.
FL_API_ACQUIRE_DATA_RDY	The *Acquisition* transaction has the final data ready.
FL_API_ENROLL_DATA_RDY	The *Enrollment* transaction has the final data ready.
FL_API_VALIDATE_DATA_RDY	The *Validation* transaction has the final data ready.
FL_API_VERIFY_DATA_RDY	The *Verification* transaction has the final data ready.
FL_API_IDENTIFY_DATA_RDY	The *Identification* transaction has the final data ready.
FL_API_END_OF_VIEW	The finger has been lifted from the sensor
FL_API_TIMEOUT	The transaction was canceled due to a timeout condition.
FL_API_IDLE_50	API has been idle for 50 milliseconds.

Synchronous Operation Implementation

The application first responds to the User's request for enrollment or identification service. It then collects the information needed for the specified operation. This may include the User's name, a User ID, the finger to process, or any other data the application requires.

The application then passes the data as parameters to the appropriate *Begin* function that in turn returns a Transaction ID that must be saved for later use. The application then polls the FAS for event messages by calling the **FLContinueTransaction** function. If an event message is available the application processes it. If no event message is available, the application simply sleeps for a short time (10 to 100 milliseconds), then calls **FLContinueTransaction** again to make another request for an event message. Event messages such as **FL_API_NEW_DISPLAY_IMAGE**, **FL_API_NEW_STATE_TEXT**, **FL_API_NEW_PROMPT_TEXT**, and so on, will be returned throughout the open operation (transaction). Eventually an **FL_API_XXXXXX_DATA_RDY** or an **FL_API_TIMEOUT** event message will be sent to indicate that the operation has completed or that the operation or has timed out due to lack of user interaction with the sensor. If the **FL_API_XXXXXX_DATA_RDY** event message is sent the application must call the balancing *End* function to gracefully end the transaction in progress. It is not necessary to call the *End* function if the transaction has been canceled by an **FL_API_TIMEOUT** event message.

Asynchronous Operation Implementation

The application first responds to the user's request for enrollment or identification service. It then collects the information needed for the specified operation. This may include the user's name, an ID, specification of the finger to process, or any other data the application requires.

The application then sets up any global information needed by the *Callback* routine. The *Callback* function's address is passed as a parameter to the appropriate *Begin* function which in turn returns a Transaction ID that must be saved for later use.

The *Begin* function starts the operation (transaction) and returns to the application where the calling thread is then free to continue with normal idle loop execution. Through the transaction process the FAS asynchronously calls the application's *Callback* function with event messages such as **FL_API_NEW_DISPLAY_IMAGE**, **FL_API_NEW_STATE_TEXT**, **FL_API_NEW_PROMPT_TEXT**, and so on. Eventually an **FL_API_XXXXXX_DATA_RDY** or an **FL_API_TIMEOUT** event message will be sent to indicate that the operation has completed or that the operation or has timed out due to lack of user interaction with the sensor.

If the **FL_API_XXXXXX_DATA_RDY** event message is sent, the application must call the balancing *End* function to gracefully end the transaction in progress. It is not necessary to call the *End* function if the transaction has been canceled by an **FL_API_TIMEOUT** event message. Once the *Callback* is executed with a message indicating the transaction has finished, it notifies the idle loop of the application (using some type of signal or global variable) that the result is waiting.

The pointer version of the *Callback* function below is prototyped in the **FLStdApi.h** header file along with the definitions for the API messages.

Asynchronous callback function:

```
void FL_Notify_Callback (uint32 uiCallerDWord, tsFL_API_MSG*pApiMsg );
```

Application Example

A complete sample application that uses the High Performance API is available in the **FLHighPerformanceAPIDemo** sub-directory. A step-by-step illustration of how to handle the *Enrollment* functionality in an asynchronous manner, is shown in the following diagram:

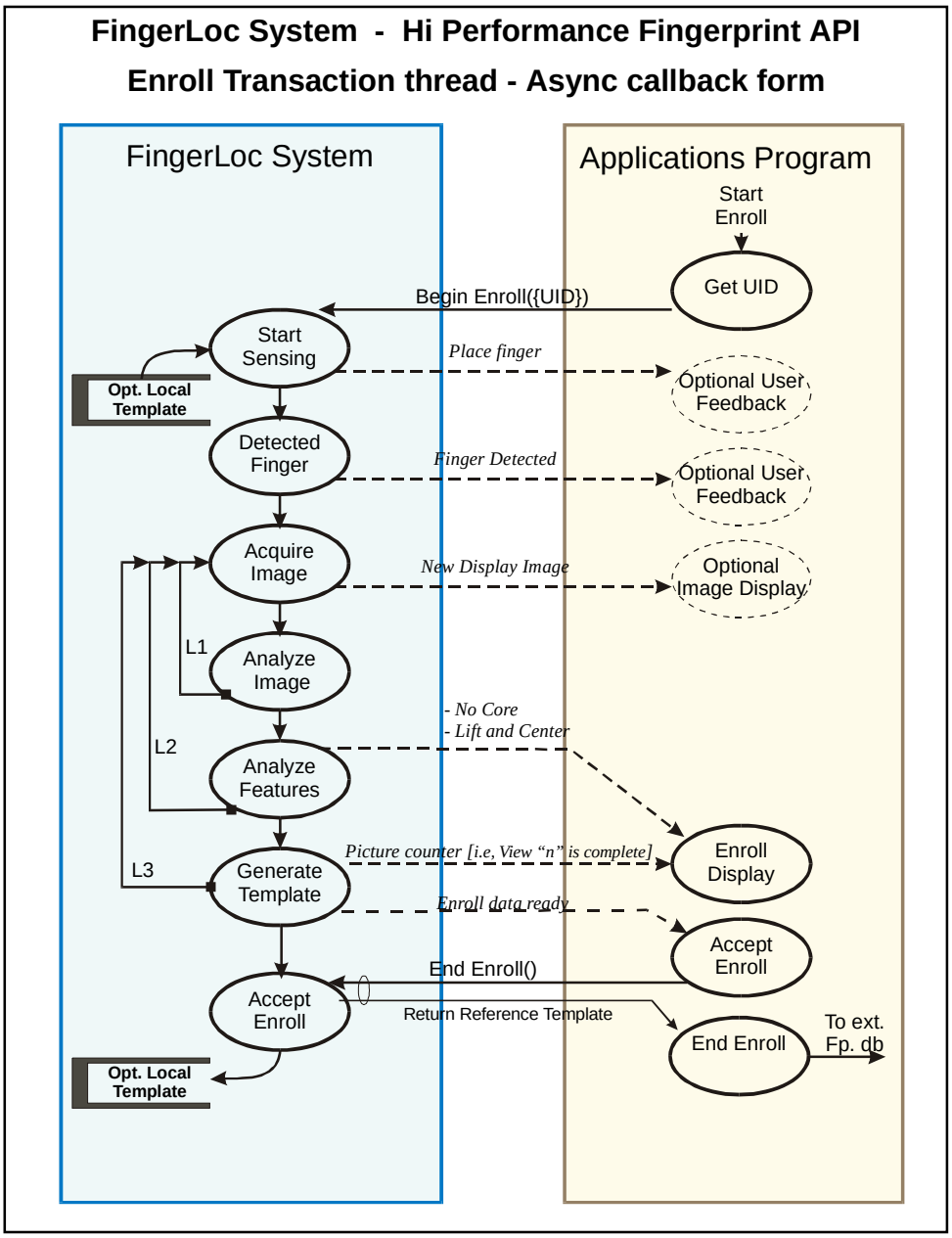


Figure 1

The Fingerprint Services API

The Fingerprint Services API provides an application with a set of functions that perform operations similar to the legacy APIs typically used with optical sensor technology. In addition, the interface maximizes the use of the TruePrint Technology Dynamic Optimization features available in the FingerLoc sensor IC during scanning operations. This interface is can be used to speed migration from existing legacy systems, or in special cases where the design of the interface is the best fit for a specific solution.

Like the High Performance API, the Fingerprint Services API requires the calling application or process to provide the user interface, if any. Unlike the High Performance API, the various operations of the Fingerprint Services API do not provide start-to-finish automatic control over primary processes such as *Enrollment*, *Verification*, *Identification*, and so on. These primary operations are accomplished in stages, where the calling application requests the FAS to perform a specific operation, or service, on a target object. Performing a combination of different steps on the target object will eventually result in the completion of a primary process.

The Fingerprint Services API provides the following functionality:

- Services to obtain sizes for objects such as images, bitmap headers, match templates, and reference templates.
- Services to acquire fingerprint images from the sensor.
- Services to create or build objects such as match templates, reference templates, and bitmap headers from acquired fingerprint image data.
- Services to compare reference template objects to match template objects in order to determine if the templates are cognate with an identical finger.

The application accomplishes *Enrollment* by requesting the FAS to acquire a fingerprint image from a finger. A reference template is created from the acquired fingerprint image by passing the image into the appropriate API function. This reference template is then stored in an external database for later use. The application is responsible for maintaining all information regarding the enrolled template.

When the application desires to identify a User it requests the API to acquire a fingerprint image for the finger that is placed on the sensor. A match template is then created from the acquired fingerprint image by passing the image into the appropriate API *Build* function. The match template is then presented as an input parameter to the API matching function along with a desired reference template. The two templates are then compared to determine if they represent the same finger. The match template is temporary - it is discarded after this comparison.

The calling application accomplishes primary matching processes such *Identification*, *Verification*, and *Validation* by presenting externally stored reference templates to the matching function of the API on a one-to-one basis with the temporary match template.

The nature of the design of the Fingerprint Services API makes it a viable solution in networked systems where an acquired match image can be collected by a remote sensor and

shipped over the network to a central processor responsible for reference template storage and matching.

Theory of Operation

Like the High Performance API, the Fingerprint Services API uses a transaction-based method to interact with the calling application. Each API operation is initiated by the application with a *Begin* function call and terminated with a balancing *End* function call. An example of such a pair is **FLAcquireImage** and **FLEndAcquireImage**. Once an operation (transaction) has begun, event messages are sent to the calling application from the FAS.

These event messages provide the calling application with feedback information as to the progress of the current transaction. The event messages contain various items such as fingerprint images that can be used for display, finger placement information, sequencing information and so on. The calling application processes these event messages in order to update user displays and to control various stages of the current transaction. The FAS sends a *Data Ready* event message to indicate that the operation is complete. The calling application processes this message and in turn calls the transaction's corresponding *End* function to obtain the final results and terminate the current transaction.

The calling application can receive the event messages from the FAS using either a synchronous or asynchronous method. The synchronous method requires the application to call into the FAS to obtain each event message. Once obtained, the application processes the event message and calls a subsequent *Release* function to release the message. This procedure is then repeated for the next event message. The asynchronous method requires the calling application to pass a pointer to a *Callback* function as a parameter to the transaction's *Begin* function call. The FAS will in turn call the *Callback* function with event messages at the time the event message is produced (asynchronous to the calling application's process execution). The application's *Callback* function provides the code necessary to process the various event messages. A call to *Release* the message must also be included in the callback procedure.

There is an advantage to using the asynchronous event message processing method instead of the synchronous event message processing method. Asynchronous event message processing allows the calling application to initiate a transaction and continue with its normal path of execution. The main application remains free for processing other tasks during the open transaction. However, it is not always possible to correctly implement the required *Callback* function necessary to enable the asynchronous method. The synchronous method requires the thread of the calling application to loop while polling for event messages during the open transaction. This method makes it more difficult for the application program to execute other tasks, such as user input, in between event message processing.

An application can prematurely terminate an in-progress transaction by calling the API **FLAbortTransaction** function from the High Performance API.

The basic operational flow for the two modes is described in the following sections.

API Function Calls

The functions in the Fingerprint Services API group are shown below.

- **FLGetImageBufferSize** - This function retrieves the buffer size required to store a scanned image that will be returned from the **FLEndAcquireImage** API function.
- **FLGetOSBmpHeaderSize** – This function retrieves the buffer size required to store a bitmap header used to describe characteristics of a displayable fingerprint image.
- **FLBuildOSBmpHeader** – This function creates a valid bitmap header by filling the contents of a pre-allocated buffer of size equivalent to that which was returned by a call to **FLGetOSBmpHeaderSize.**
- **FLBeginAcquireImage** – This function initiates an operation (transaction) to acquire a fingerprint image for a finger placed on the sensor.
- **FLEndAcquireImage** – This function retrieves the fingerprint image from an *Acquire Image* operation once the FAS signals that the results are available.
- **FLGetMatchTemplateSize** – This function retrieves the buffer size required to store a match template.
- **FLBuildMatchTemplate – This function c**reates a match template for the fingerprint described by an image returned by **FLEndAcquireImage**. The resulting fingerprint template is stored in a pre-allocated buffer of size equivalent to that which was returned by a call to **FLGetMatchTemplateSize.**
- **FLGetReferenceTemplateSize** – This function retrieves the buffer size required to store a large reference template.
- **FLGetSmallReferenceTemplateSize** – This function retrieves the buffer size required to store a reduced size, or small, reference template.
- **FLConvertNormalReferenceTemplateToSmall** – This function converts a normal, or large reference template to a reduced size, or small, reference template.
- **FLBuildReferenceTemplate** – This function creates a reference template for the fingerprint described by an image returned by **FLEndAcquireImage**. The resulting fingerprint template is stored in a pre-allocated buffer of size equivalent to that which was returned by a call to **FLGetReferenceTemplateSize**.
- **FLMatchTemplatePair**
  Compares a reference template to a match template and returns a result to indicate that the two templates match the same finger.

- **FLMatchTemplatePairEx** – This function compares a reference template to a match template and returns a result to indicate that the two templates match the same finger. In addition the function returns the quality scoring values that were used to make the match.

- **FLReleaseMessage** - This function releases (deletes) the memory space allocated by a valid message that was returned by a previous call to **FLContinueTransaction** while operating in synchronous mode or a message that was received by a callback function while operating in asynchronous mode.

- **FLAbortTransaction** – This function aborts the current operation (transaction) and returns the FAS to an idle state. Calling this function will abort any operation (transaction) currently in progress. This function requires the transaction ID that was returned by the operation originally initiated with a *Begin* function call.

The functions in this group are listed alphabetically and described in detail in the "API Reference" section of this manual.

Event Messages

Event messages are passed throughout the system using a pointer to a **tsFL_API_MSG** event message structure as shown below. The first member of the event message structure contains the message type. The second parameter contains the sub-message number for the respective message type. Only a few message types actually have a sub-message number. The third parameter contains a pointer to the actual message data. This data depends upon the message type. For example, if the message type is an **FL_API_NEW_DISPLAY_IMAGE**, the **pMessageData** pointer points to a buffer that contains a raw fingerprint image that can be formatted and displayed by the calling application. Details of the message types, sub-message numbers and message data information can be found in the header file **FLStdPI.h**.

The table below outlines the various message types that can be expected when processing event messages.

```
typedef struct
{
uint16  uiMessageType ;
uint16  uiSubMessageNum ;
void*   pMessageData ;
} tsFL_API_MSG;
```

MESSAGE TYPE	DESCRIPTION
FL_API_NEW_DISPLAY_IMAGE	The sensor has a new image to be displayed.
FL_API_CLEAR_DISPLAY_IMAGE	Clear the last displayed image.
FL_API_NEW_STATE_TEXT	The state of the sensor has changed - display a new text message.
FL_API_CLEAR_STATE_TEXT	The sensor state message text is no longer valid.
FL_API_PLAY_PROMPT_SOUND	Play sound for action done, end of capture, capture reject, and so on.
FL_API_NEW_PROMPT_TEXT	Prompt the user for a finger action.
FL_API_CLEAR_PROMPT_TEXT	Remove any user prompts from the display.
FL_API_ACQUIRE_DATA_RDY	The *Acquisition* transaction has the final data ready.
FL_API_ENROLL_DATA_RDY	The *Enroll* transaction has the final data ready.
FL_API_VALIDATE_DATA_RDY	The *Validation* transaction has the final data ready.
FL_API_VERIFY_DATA_RDY	The *Verification* transaction has the final data ready.
FL_API_IDENTIFY_DATA_RDY	The *Identification* transaction has the final data ready.
FL_API_END_OF_VIEW	The finger has been lifted from the sensor.
FL_API_TIMEOUT	The transaction was canceled due to a timeout.
FL_API_IDLE_50	API has been idle for 50 milliseconds.

Synchronous Operation Implementation

The application first responds to the user's request for *Enrollment* or *Identification* services. It then collects the information needed for the specified operation. This may include the user's name, an ID, the finger to process, or any other data the application requires.

The application then passes the data as parameters to the appropriate *Begin* function which in turn returns a Transaction ID that must be saved for later use. The application then polls the FAS for event messages by calling the **FLContinueTransaction** function. If an event message is available the application processes it. If no event message available the application simply sleeps for a short duration (10 to 100 milliseconds) and then calls **FLContinueTransaction** again to make another request for an event message.

Event messages such as **FL_API_NEW_DISPLAY_IMAGE**, **FL_API_NEW_STATE_TEXT**, **FL_API_NEW_PROMPT_TEXT**, and so on, will be returned throughout the open operation (transaction). Eventually an **FL_API_XXXXXX_DATA_RDY** or an **FL_API_TIMEOUT** event message will be sent to indicate that the operation has completed or that the operation or has timed out due to lack of user interaction with the sensor. If the **FL_API_XXXXXX_DATA_RDY** event message is sent the application must call the balancing *End* function to gracefully end the transaction in progress. It is not necessary to call the *End* function if the transaction has been canceled by an **FL_API_TIMEOUT** event message.

Asynchronous Operation Implementation

The application first responds to the User's request for *Enrollment* or *Identification* services. It then collects the information needed for the specified operation. This may include the user's name, an ID, specification of the finger to process, or any other data the application requires.

The application then sets up any global information needed by the *Callback* routine. The *Callback* function's address is passed as a parameter to the appropriate *Begin* function which in turn returns a Transaction ID that must be saved for later use.

The *Begin* function starts the operation (transaction) and returns to the application where the calling thread is then free to continue with normal idle loop execution. Through the transaction process the FAS asynchronously calls the application's callback function with event messages such as **FL_API_NEW_DISPLAY_IMAGE**, **FL_API_NEW_STATE_TEXT**, **FL_API_NEW_PROMPT_TEXT**, and so on.

Eventually an **FL_API_XXXXXX_DATA_RDY** or an **FL_API_TIMEOUT** event message will be sent to indicate that the operation has completed or that the operation or has timed out due to lack of user interaction with the sensor. If the **FL_API_XXXXXX_DATA_RDY** event message is sent the application must call the balancing *End* function to gracefully end the transaction in progress. It is not necessary to call the *End* function if the transaction has been canceled by an **FL_API_TIMEOUT** event message. Once the *Callback* is executed with a message indicating the transaction has finished, it notifies the idle loop of the application (using some type of signal or global variable) that the result is waiting.

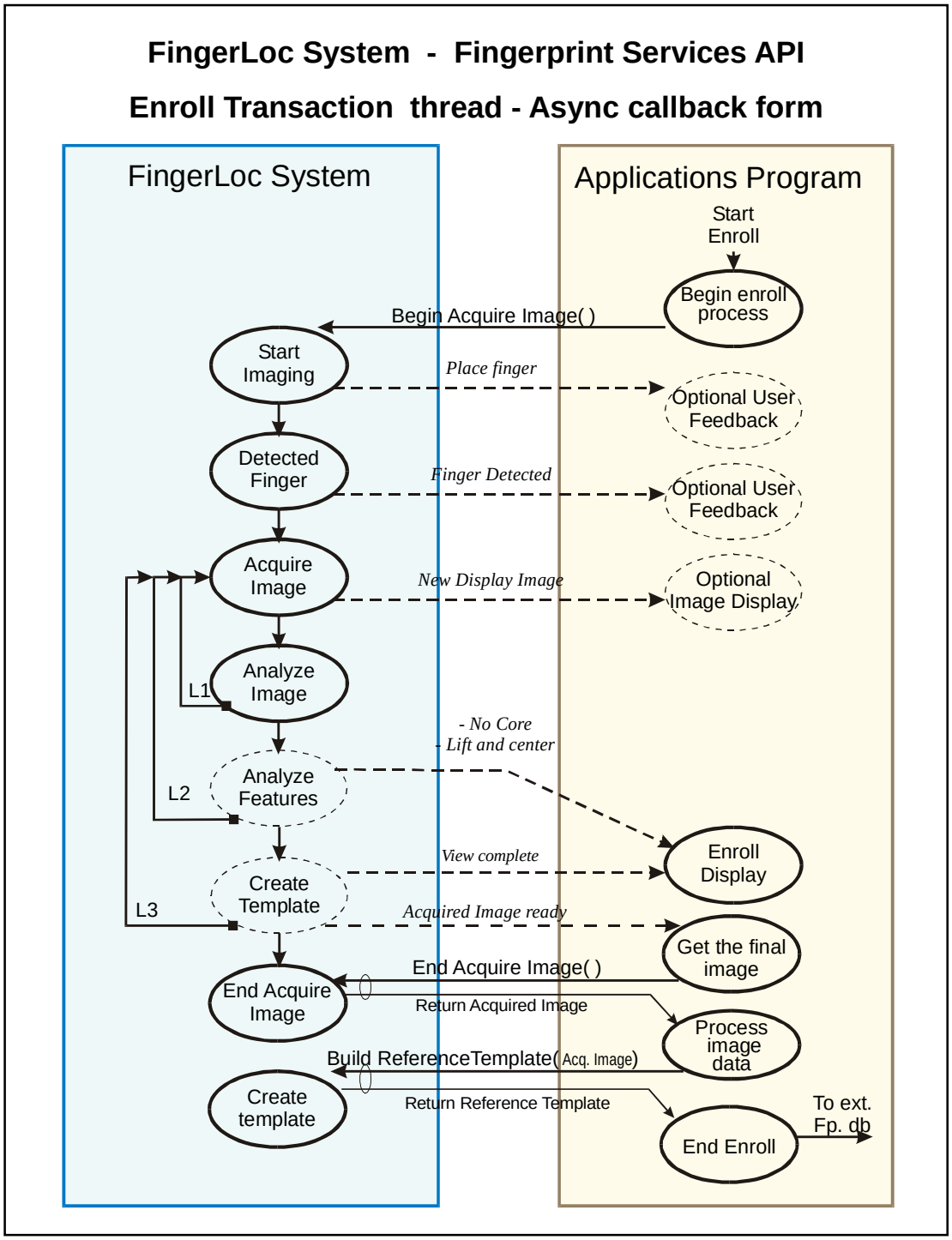
The pointer version of the *Callback* function below is prototyped in the **FLStdApi.h** header file along with the definitions for the API messages.

Asynchronous callback function:

```
void FL_Notify_Callback (uint32 uiCallerDWord, tsFL_API_MSG*pApiMsg );
```

Application Example

A complete sample application that uses the Fingerprint Services API is available in the **FLServicesAPIDemo** sub-directory. A step-by-step illustration of how to handle the *Enrollment* functionality in an asynchronous manner, is shown in the following diagram:



The Convenience API

The Convenience API is an implementation in which the FAS provides it's own Graphical User Interface objects, such as feedback windows and interactive user dialog box controls, during primary operations such as *Enrollment*, *Identification*, and so on. For this reason, the Convenience API allows a means to quickly construct demonstrations directly from applications in order to evaluate the FingerLoc sensor IC.

Most developers will want to replace the functionality provided by this group of API functions with that of either the High Performance API or the Fingerprint Services API for the final product.

The Convenience API is designed to provide a calling application with a suite of functions that will maximize use of the Dynamic Optimization features available in the FingerLoc sensor IC during finger scanning operations. Additionally, the Convenience API provides start-to-finish automated control of the primary processes, such as *Enrollment*, *Verification*, and *Identification* once the process has been initiated by the calling application. The Convenience API provides the following functionality:

- **Enrollment** (creation of reference templates) of a User's fingers.
- **Identification** to look up which, if any, User stored in the internal database belongs to a finger on the FingerLoc sensor IC.
- **Validation** that the finger on the sensor belongs to a specified User according to the specified User's template stored in the internal database.
- **Verification** that a finger on the FingerLoc sensor IC matches that of a fingerprint template stored in an external template passed in.

The automation features of the Convenience API makes it unnecessary for the calling application to manipulate any type of low level sensor objects such as collecting raw fingerprint images, or creating reference or match templates. Fingerprint templates created by the Convenience API *Enrollment* process are always stored in the internal FingerLoc database, and additionally, exported to the calling application for optional external storage. *Identification*, *Validation*, and *Verification* operations can be called once a finger has been enrolled.

The application accomplishes *Enrollment* simply by calling the enroll function of the API and passing it the desired user and finger ID information. The FAS automatically displays a window on the computer display that guides the user through the enroll process. Fingerprint images produced by the scanning process are displayed to the user on a small child window embedded inside the larger interface screen. Control is returned to the calling application when the actual enroll API interface function returns. Unlike the High Performance API and the Fingerprint Services API, the Convenience API does not supply the calling application with any type of event messages. The result of an operation is determined by the value returned by the called function.

Identification, validation, and verification are accomplished in a similar manner. The application simply calls the desired API function while passing the necessary information and the FAS

automatically executes the entire operation. The FAS displays and controls all necessary user interface mechanisms.

The Convenience API does not provide options to customize the user interfaces and should only be used where it's design would be beneficial to meeting the goals of the solution.

Theory of Operation

Unlike the High Performance and Fingerprint Services API, the Convenience API is not transaction-based. As described in the preceding section, operations are initiated by calling the appropriate function in the API. Once initiated, control is never returned to the calling application until the operation is complete and the function returns.

Source code in the sample test program, **FingerLocAPIDemo** demonstrates usage of the Convenience API.

API Function Calls

The functions in the Convenience API group are shown below.

- **FingerLocEnroll** – This function causes the FAS to execute a complete process to enroll the desired finger for a User.
- **FingerLocIdentify** – This function causes the FAS to execute a complete process to look up which, if any, User stored in the internal database belongs to a finger on the sensor.
- **FingerLocValidateID** – This function causes the FAS to execute a complete process to confirm that the User that belongs to the finger placed on the FingerLoc sensor IC matches the user with the same user ID that is stored in the internal database.
- **FingerLocValidateFingers** - This function causes the FAS to execute a complete process to confirm that the finger placed on the FingerLoc sensor IC matches the fingerprint template passed into the function from an external database.

> **Note:** Each of the Convenience API functions require a special handle to be passed as a parameter. This handle is obtained by calling the **FingerLocInit** function as described in the "Utility and Common API" section of this manual.

The functions in this group are listed alphabetically and described in detail in the "API Reference" section of this manual.

The Image Capture API

The functions of the Image Capture API provide support for applications that have their own built-in fingerprint matching capability. The API provides several functions that allow the calling application to acquire a raw fingerprint image from the FingerLoc sensor IC in addition to obtaining information such as the presence of a finger on the sensor. The Image Capture API does not make use of any Dynamic Optimization capabilities of the FingerLoc sensor IC. As a result, fingerprint images returned by the Image Capture API will be generally inferior in quality to those returned using the *Image Acquisition* functions of the Fingerprint Services API. The **FLBeginAcqureImage** - **FLEndAcquireImage** function pair of the Fingerprint Services API should be used in most situations. This API is included for backward compatibility.

API Function Calls

The functions in the Image Capture API group are shown below.

- **FingerLocOpenStream** – This function instructs the FAS to enable imaging mode. Fingerprint images will be available whenever a finger is placed on the FingerLoc sensor IC. Retrieve the actual fingerprint images by calling the **FingerLocGetCurrentImage** function. This function can not be used simultaneously when making calls to the High Performance API or the Fingerprint Services API.

- **FingerLocCloseStream** – This function instructs the FAS to disable imaging mode. This function can not be used simultaneously when making calls to the High Performance API or the Fingerprint Services API.

- **FingerLocGetCurrentImage** – This function retrieves a fingerprint image from the FAS when an image is available. Use the **FingerLocGetImageState** function to determine that a finger has been placed on the FingerLoc sensor IC and that a valid image is available for retrieval. The image is placed into a buffer that is allocated by this function. This buffer must be destroyed using the **FingerLocReleaseImage** function when finished with the image. This function can not be used simultaneously when making calls to the High Performance API or the Fingerprint Services API.

- **FingerLocReleaseImage** – This function releases the buffer created by **FingerLocGetCurrentImage**. Call this function when finished using the image data. This function can not be used simultaneously when making calls to the High Performance API or the Fingerprint Services API.

- **FingerLocGetImageState** – This function returns information that indicates if there is a finger on currently on the FingerLoc sensor IC and that there is a valid image available for retrieval.

Note: Each of the Image Capture API functions require a special handle to be passed as a parameter. This handle is obtained by calling the **FingerLocInit** function as described in the “Utility and Common API” section of this manual.

The functions in this group are listed alphabetically and described in detail in the “API Reference” section of this manual.

The Utility and Common API

The Utility and Common API provides the programmer with several commonly needed utility functions. Functions for image re-sampling, image format conversion, database manipulation, program initialization, sensor communication control, and error reporting are included.

API Function Calls

The functions in the Utility and Common API group are shown below.

- **FingerLocResample150** – This function magnifies a square input image by a factor of 1.5. For example, a 128 x 128-pixel input image results in a 192 x 192-pixel image.
- **FingerLocResample200** – This function magnifies a square input image by a factor of 1.5. For example, a 128 x 128-pixel input image results in a 192 x 192-pixel image.
- **FingerLocAllocOSBitmap** – This function allocates a block of buffer memory the size of a Windows bitmap information structure. The buffer is then filled with the proper information necessary to describe a 144 x 144-pixel bitmap, including the RGB look-up table.
- **FingerLocDeleteOSBitmap** – This function destroys a memory buffer allocated with the **FingeLocAllocOSBitmap** function.
- **FingerLocConvertRawImageToOSBmp** – This function converts a raw eight-bit gray scale 128 x 128-pixel raw scanned image into an eight-bit gray scale 144 x 144-pixel Windows bitmap.
- **FingerLocGetLeds** – This function retrieves the state of each of the four user-programmable outputs located on the FingerLoc sensor IC (each capable of driving an LED).
- **FingerLocSetLeds** – This function sets the state of any one of four general-purpose user-programmable outputs located on the sensor (each capable of driving an LED).
- **FLGetCommPort** – This function retrieves information regarding the current hardware port settings used to communicate with the sensor.
- **FLReadCommPort** – This function reads a stream of bytes from the communication port.
- **FLWriteCommPort** – This function writes a stream of bytes directly to the communication port.
- **FLReleaseCommPort** – This function releases control of the serial communications port.

- **FLRestartCommPort** – This function instructs the FingerLoc server to restart the communication port using the current settings.

- **FingerLocGetVersion** – This function retrieves the type and version of the sensor, FingerLoc server, and associated FingerLoc DLLs.

- **FingerLocInit** – This function initializes the FAS and returns a context handle that is a required parameter with various functions. The context handle returned from this function is only required by a few API operations. Do not call this function unless a handle is actually required.

- **FingerLocClose** – This function closes an initialized FAS originally initiated with the **FingerLocInit** function.

- **FingerLocGetNextUser** – This function retrieves the information regarding the next User stored in the internal FAS database. The function can either start from the beginning of the database or from a specified User. Call this function in a loop to build a list of Users ID and associated Display Names.

- **FingerLocGetResultDetails** – This function retrieves information regarding the last error that occurred in the FAS. Call this function in response to an API function failure. The error code is cleared when this function is called, but that does not mean that the actual error condition that caused the fault will be cleared. A list of error codes is found in the file **FLStdApi.h**.

The functions in this group are listed alphabetically and described in detail in the “API Reference” section of this manual.

Event Messages, Error Codes, and Return Codes

This section describes the messages and return value enumerations in use in this version of the API. The latest defines are found in the **FLStdApi.h** header file.

Event Messages

Event messages are returned when using either the High Performance or Fingerprint Services APIs (see the respective sections within this manual for detailed information regarding the APIs). The **FLStdApi.h** contains details regarding the various API event messages and their usage.

Error codes

All error codes are negatively numbered. If the return received from either an API call or from the **FingerLocGetResultDetails** function is negative, this indicates that there has been an error somewhere in the FAS. Refer to the **FLStdApi.h** file for up-to-date definitions of error codes. The errors are listed below.

```
FL_FUNCTION_FAILED            // Generic error
FL_INITALIZE_FAILED           // Startup error
FL_OUT_OF_MEMORY              // Could not allocate memory block
FL_INVALID_CONTEXT            // Context handle is invalid
FL_COMM_ERROR                 // Communications error
FL_OS_ERROR                   // Operating system error
FL_INVALID_KEY_SIZE
FL_UNSUPPORTED_IMAGE_RESOLUTION
FL_INVALID_IMAGE_RESOLUTION
FL_INVALID_FRAME_RATE
FL_INVALID_ARGUMENT
FL_FUNCTION_NOT_SUPPORTED
FL_BAD_HANDLE
FL_BAD_POINTER
FL_INVALID_CRC
FL_STREAM_CLOSED
FL_STREAM_OPENED
FL_HEAP_BAD
FL_STACK_UNDERFLOW
FL_DATASTORE_ERROR
FL_FILE_BUSY
```

Return Codes

Return codes are positively numbered. These codes are used to indicate the results of the various API calls. Refer to the **FLStdApi.h** file for up-to-date definitions of error codes. Refer to the various API function calls for details on what the various return codes mean when used within the respective functions.

```
FL_OK                  // function succeeded
FL_UNKNOWN             // desired user not found in internal database
FL_MATCH               // the finger on the sensor matches
FL_NO_MATCH            // the finger on the template does not match
FL_USER_ALREADY_ENROLLED
FL_NO_IMAGE
FL_USER_CANCELED
FL_MAX_USER_FINGERS_ENROLLED
FL_IMAGE_RDY
FL_BAD_IMAGE
FL_TIMEOUT
FL_FILE_NOT_FOUND
FL_FINGER_PRESENT
```

The High Performance API Demonstration

An application called the **FLHighPerformanceAPIDemo** is delivered with the Software Developer's Kit. Not surprisingly, it demonstrates the use of the FingerLoc High Performance API. It exercises the major operations provided by the API in both the polled and callback modes that are described elsewhere in this document.

An executable demonstration application and full C++ source code for the demonstration are provided. The source code provides an example of how the High Performance API calls are used together to enroll or match fingerprints in the FingerLoc database.

Description

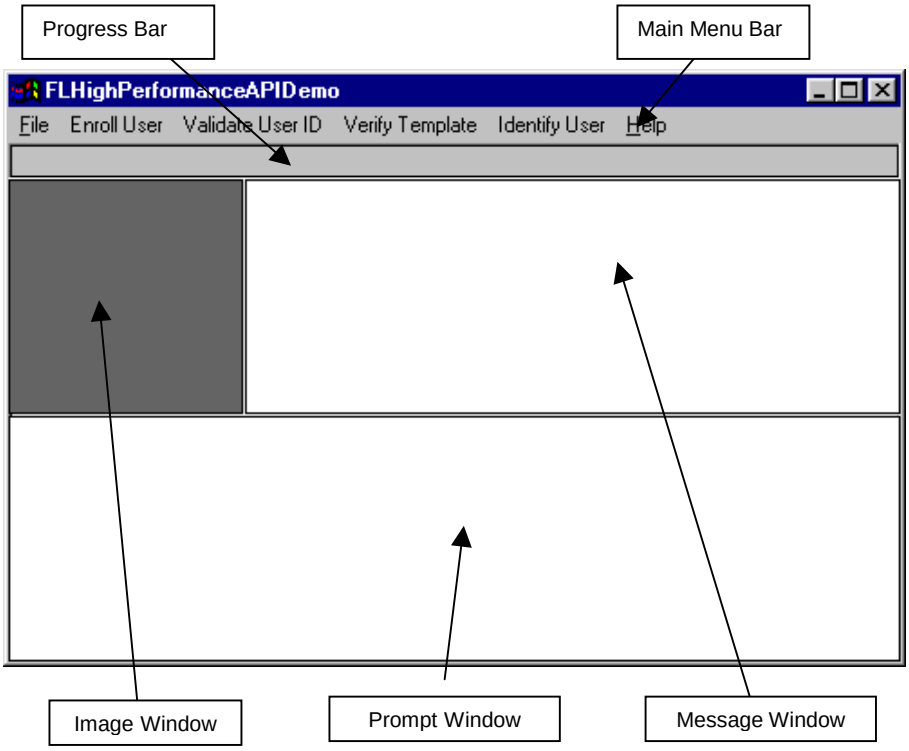
Six options are provided on the Main Menu Bar:

File	
Exit	Close and return to the desktop.
Enroll User	
Polled	
Callback	
Validate User ID	
Polled	
Callback	
Verify Template	
Identify User	
Help	
About…	Display copyright information about *FingerLoc Aware.*

- The **File** option allows the application to be terminated, just as if selecting the close box on the title bar.
- The **Enroll User** option allows a User to be placed in the FingerLoc database.
- The **Validate User ID** option provides a one-to-one match with a User record in the database.
- The **Verify Template** option provides a one-to-one match between the last enrolled finger and the next finger placed on the FingerLoc sensor IC.
- The **Identify User** option provides a "one-to-many" match of the next finger placed on the FingerLoc sensor IC against all fingers in the FingerLoc database.
- The **Help** option shows the standard Help dialog box.

The Main Window contains four sections. Just below the Main Menu Bar, is a Progress Bar that indicates the progress of an *Enrollment* process. To the left and below the Progress Bar is a gray Image Window that displays images from the sensor. To the right of the Image Window

is a Message Display Window that displays informational messages. Across the bottom of the Main Window is the Prompt Window that displays instructional messages to aid a user in finger positioning during enrollment and matching. The following paragraphs describe the use of each option in more detail.



File

Selection of the **File** option allows only one subsequent operation, **Exit** from the program.

Enroll User

Select the **Enroll User** option to see a pull-down list with two selections, “Polled” and “Callback”. From the perspective of someone using **FLHighPerformanceAPIDemo** both methods appear to work the same. Exact differences will be detailed later during implementation discussions.

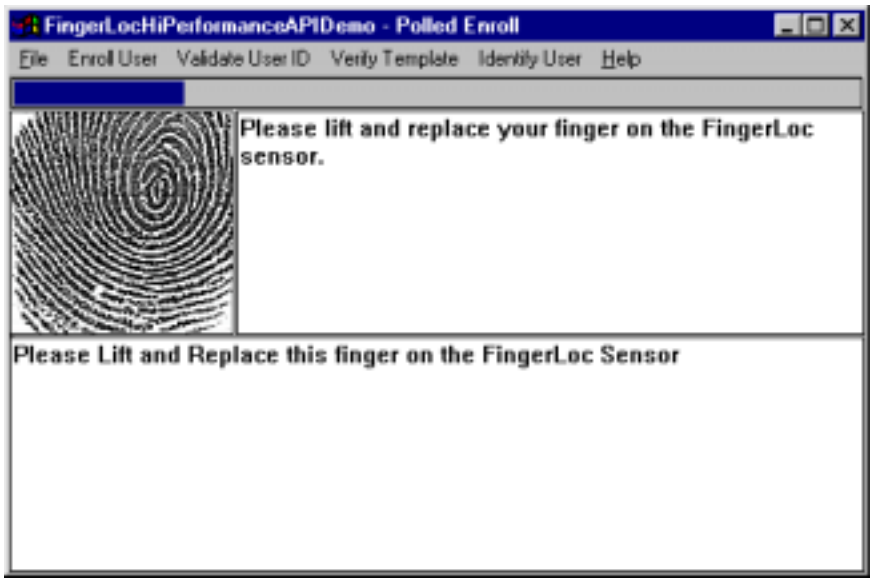
When an *Enrollment* is initiated, a name for the user and a finger specification is required. The following illustration shows the **User Identification** dialog box that is used to collect this information.

Type a unique User Name in the **Name** text field, then click an adjacent button to specify the finger to be enrolled. Click **OK** to continue the enrollment, or **Cancel** to end the process.

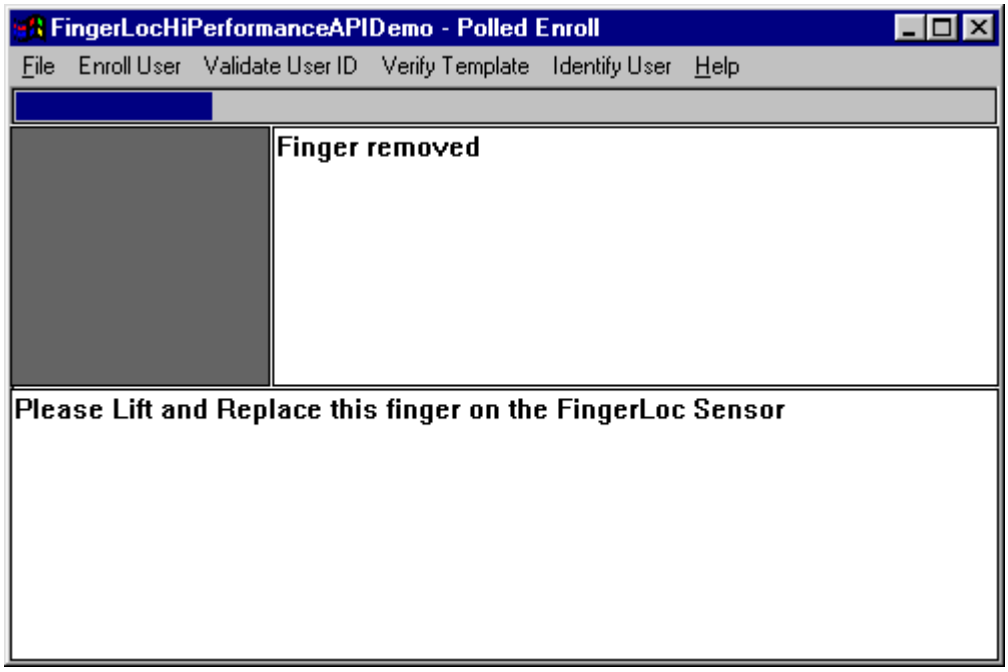
Figure 6 shows the continuation of the *Enrollment* process with the demonstration application prompting the user to place the selected finger on the FingerLoc sensor IC. Of course, it will

accept a different finger, but the finger on the sensor will be permanently identified in the database as the one selected.

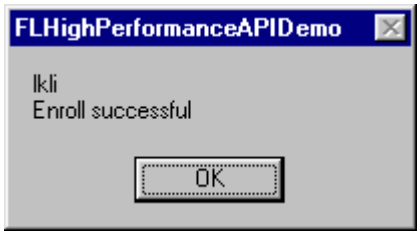
Place a finger on the FingerLoc sensor IC to capture an image and evaluate it. The image is displayed, a message and a prompt are displayed, and the progress indicator is updated.



Removing the finger from the FingerLoc sensor IC updates the windows again indicating that **FLHighPerformanceAPIDemo** has detected that the finger has been removed and still prompts the user to place the finger back on the sensor as shown in Figure 5

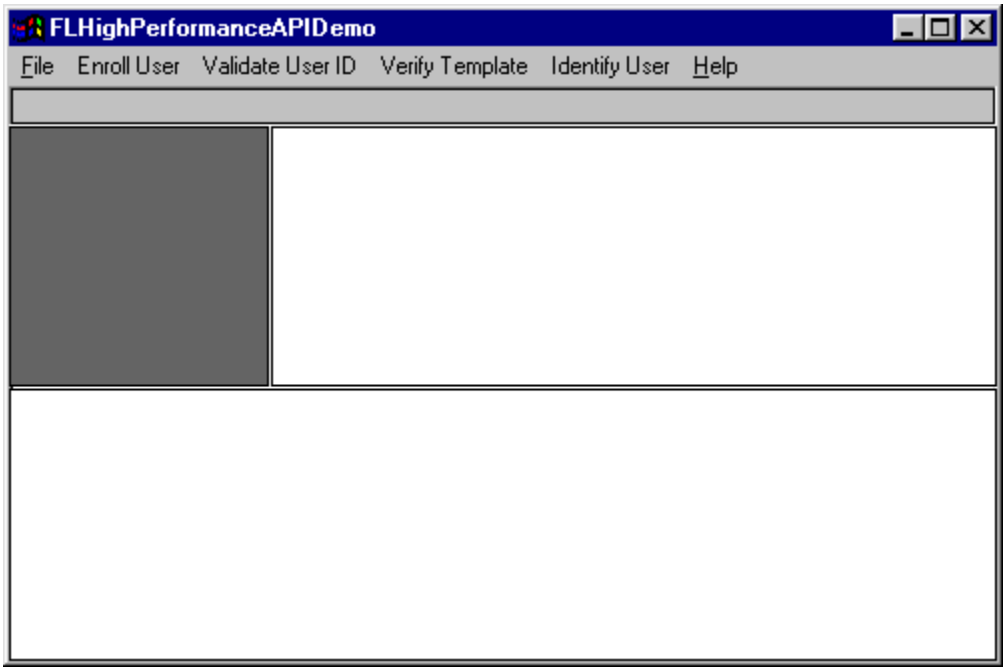


A total of three views of the finger will be taken, from which only the one with the best characteristics will be saved in the database. If the enrollment is successful, the dialog box shown in Figure 6 is displayed (the lkli is replace with the user name), thus completing the enrollment.

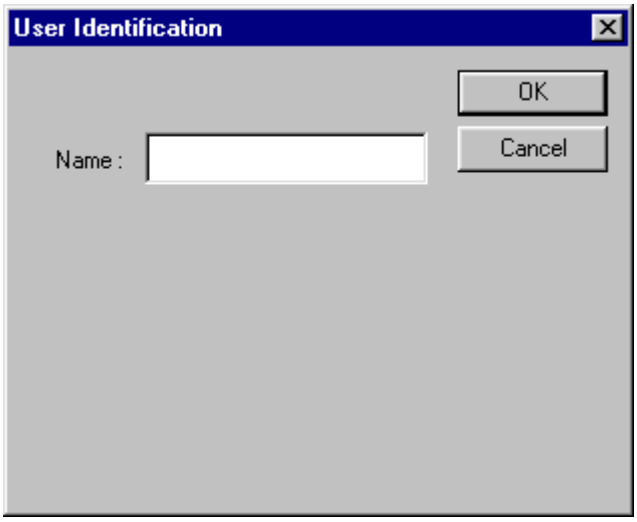


Validate User ID

Selection of the **Validate User ID** option creates a pull-down list with two selections, **Polled** and **Callback**. From the perspective of someone using **FLHighPerformanceAPIDemo**, both methods appear to work the same. Exact differences will be detailed later during implementation discussions.



Once the running method (**Polled** or **Callback**) is selected, the **User Identification** dialog box is displayed to select which User Id is to be validated. Once the **Name** has been entered, a single view of the fingerprint is taken.



If the template created from the fingerprint matches the template from the database for the selected User Id the user access is accepted, otherwise the user access is rejected.

Verify Template

Selection of the Verify Template option creates a pull-down list with two selections, Polled and Callback. From the perspective of someone using **FLHighPerformanceAPIDemo** both methods appear to work the same. After the running method is selected, the Verify Template option performs a one-to-one match of the last enrollment template created from the **Enroll User** option against a template created from the next fingerprint seen. The results of the matching process are the same as for the **Validate User Id**, acceptance or rejection.

Identify User

Selection of the Identify User option creates a pull-down list with two selections, “Polled” and “Callback”. From the perspective of someone using **FLHighPerformanceAPIDemo** both methods appear to work the same. After the running method is selected, the Identify User option performs a many-to-one match of all users that have been saved in the Datastore against a template created from the next fingerprint seen. The result of the matching process is the same as for the **Validate User Id**, acceptance or rejection.

Help

The Help option displays the standard Windows application Help dialog box.

Implementation

The **FLHighPerformanceAPIDemo** application is built using the Microsoft® Visual Studio version 6.0 Development System. It uses MFC without the Document/View architecture. The source code for this program is provided with the Software Developer's Kit.

FLHighPerformanceAPIDemo acts as the interface between the user and the TruePrint Server. Communication with the server is handled by calls through the **FLRpc.dll**. The server performs the control and sequencing for each operation. **FLHighPerformanceAPIDemo** displays image processing messages and user prompts and informational text messages generated by the server. It monitors the state of each operation to determine what should be done next.

The **FLHighPerformanceAPIDemo** can perform each operation in Polled or in Callback mode. The Polled mode code for each operation is much like the following code that conducts an enrollment. The call to **FLBeginEnroll()** notifies the server that an enrollment process is starting. The server conducts the enrollment by monitoring the number of times a finger has been placed on the sensor.

The server relays images, prompt, and informational messages to **FLHighPerformanceAPIDemo** through the call to **FLContinueTransaction()**. Each call to **FLContinueTransaction()** retrieves one message. While the operation is not complete and a timeout has not occurred and the user has not cancelled the operation, call **DoProcessFLMessage()** to process the last message retrieved. If a message was retrieved, process it otherwise pause to allow other processes to have CPU. If the last message received was one that indicates that the operation is complete and that the results of the operation are ready, do not process any more messages (**bImageAvailable** will be set by **DoProcessFLMessage()** so the While loop will terminate).

```
uiCurrentID = FLBeginEnroll(  NULL,
        uiCallerDWord,
        &MatchingCriteria,
        &UserId,
                                    UserInfo.m_iFinger,
        &uiMaxEnrollTemplateSize);

while( !bDone && !bTimeOut && !bImageAvailable )
{
        uint16 iRetMsgId = DoProcessFLMessage(Msg);
        if      (       iRetMsgId != FL_API_ENROLL_DATA_RDY &&
                        iRetMsgId != FL_API_VALIDATE_DATA_RDY &&
                        iRetMsgId != FL_API_VERIFY_DATA_RDY &&
                        iRetMsgId != FL_API_IDENTIFY_DATA_RDY &&
                        !bDone)
        {
                // If no message wait a bit and try again
                Msg = FLContinueTransaction(uiCurrentID);
                if( Msg == NULL)
                        Sleep(100);
        }
}
```

The *Callback* approach passes the address of the *Callback* routine to the **FLRpc.dll** when an operation is initiated.

```
uiCurrentID = FLBeginEnroll(  DoAsyncCallBack,
                                        uiCallerDWord,
                                        &MatchingCriteria,
                                        &UserId,
                                        UserInfo.m_iFinger,
                                        &uiMaxEnrollTemplateSize);
pseudoThis = this;
```

The Callback routine calls **DoProcessFLMessage()** to process each message that **FLRpc.dll** passes to it.

> **Note:** The *Callback* function is not a member of any class because that would violate C++ conventions. Therefore it does not have access to the C++ **this** pointer
>
> To allow the *Callback* routine to call a function that is a class member, a static variable named **pseudoThis** is set to the C++ **this** pointer. A reference to the **pseudoThis** pointer allows the *Callback* to call a member function **DoProcessFLMessage()**;

```
static void DoAsyncCallBack(unsigned long val,tsFL_API_MSG *Msg)
{
       if (pseudoThis != (CMainFrame *)NULL)
              pseudoThis->DoProcessFLMessage(Msg);
}
```

The heart of the **FLHighPerformanceAPIDemo** is the function **DoProcessFLMessage()**. It processes the messages returned from the server for all operations.

```
switch(uiMessageType)
{
case FL_API_NEW_DISPLAY_IMAGE:
               // new fingerprint image is available so display it in the
               // window
               break;
case FL_API_CLEAR_DISPLAY_IMAGE:     // Clear the last displayed scan
               // clear the image window
               break;
case FL_API_NEW_STATE_TEXT:
               // set new text in the state window
               break;
case FL_API_CLEAR_STATE_TEXT:
               // clear the state window
               break;
case FL_API_PLAY_PROMPT_SOUND:
               // Play sound for action done, end of capture, capture
               //reject, etc.
               // play a sound - pMessageData contains name of file to
               //play
               break;
case FL_API_NEW_PROMPT_TEXT:
               // place new prompt in prompt window
               break;
case FL_API_CLEAR_PROMPT_TEXT:
               // clear prompt window
               break;
case FL_API_ACQUIRE_DATA_RDY:
case FL_API_ENROLL_DATA_RDY:
case FL_API_VALIDATE_DATA_RDY:
case FL_API_VERIFY_DATA_RDY:
case FL_API_IDENTIFY_DATA_RDY:
// The current transaction has the final data ready.
               // mark data available
               break;
case FL_API_END_OF_VIEW:
               // during enroll, process end of view messages and display
               //prompt and state messages
               switch(Msg->uiSubMessageNum)
               {
               case FL_API_NO_CORE:
                      break;
               case FL_API_LIFT_AND_CENTER:
                      break;
               case FL_API_LIFT_AND_REPLACE:
                      break;
               case FL_API_LAST_VIEW:
                      break;
               }
               break;
case FL_API_TIMEOUT:
               break;
}
```

As can be seen from the preceding code fragment, there are both main and a sub-message types. The main message type is used by the server to signal to **FLHighPerformanceAPIDemo** that it must perform some action on the behalf of the server. These actions include displaying an image or prompting the user to reposition a finger on the sensor in response to these messages.

The sub-message is used during an enrollment operation to signal what user message needs to be presented to the user to get the user to place a finger on the FingerLoc sensor IC and remove the finger from the sensor a number of time times. The multiple views taken give the server a much better possibility of a later match.

The remainder of the code that makes up the **FLHighPerformanceAPIDemo** is standard MFC code and glue and will not be further described.

API Reference

This section contains detailed information on the available API functions in the FAS. This material is arranged in alphabetical order by function name.

FingerLocAllocOSBitmap

The Utility and Common API

FingerLocAllocOSBitmap(
int16 iPixelCount)

FingerLocAllocOSBitmap creates and initializes a Device-Independent Bitmap (DIB). The memory for the DIB is allocated, the header and the color lookup table are filled out and the pixel values initialized to zero. A DIB begins with a **BITMAPINFOHEADER** structure followed by a color lookup table, which is then followed by image data.

The function **FingerLocDeleteOSBitmap()** is called to release the memory allocated by this function.

Parameters

PARAMETER	DESCRIPTION
iPixelCount	The dimension in pixels of the image data. The image is assumed to be square.

Returns

A pointer to allocated bitmap upon success, NULL upon error.

FingerLocClose

The Utility and Common API

```
FingerLocClose(
                FL_CONTEXT_HANDLE ContextHnd)
```

FingerLocClose closes the FingerLoc sub-system that was previously opened by a call to the **FingerLocInit()** function.

Parameters

PARAMETER	DESCRIPTION
ContextHnd	The context handle originally returned by **FingerLocInit()**.

Returns

FL_OK	The operation was successful.
FL_INVALID_CONTEXT	The operation did not complete successfully - the handle passed was not correct.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocCloseStream

FingerLocCloseStream(
FL_CONTEXT_HANDLE ContextHnd)

FingerLocCloseStream terminates the **Image Capture** operation started by **FingerLocOpenStream**.

Parameters

PARAMETER	DESCRIPTION
ContectHnd	The context handle returned by **FingerLocInit()**

Returns

FL_OK	Normal return, system is no longer in image streaming mode.
FL_STREAM_CLOSED	The image stream was already closed.
FL_INVALID_CONTEXT	Operation failed - the context handle was invalid
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocConvertRawImageToOSBmp

The Utility and Common API

```
FingerLocConvertRawImageToOSBmp(
                                    BITMAPINFO* pNewDib
                                    uint8* pPixelData,
                                    uint16 iImageSize)
```

FingerLocConvertRawImageToOSBmp moves a linear block of pixels, one byte per pixel, into a bitmap object. It is assumed that the upper **BITMAPINFO** portion of the bitmap object is filled in and describes the output image. Use **FingerLocAllocOSBitmap()** to allocate and set up the correct bitmap object. If the output image is larger than the input image, the input image is centered within a border of zeroes. The rows of the input image are written to the output image in reverse order.

Parameters

PARAMETER	DESCRIPTION
pNewDib	A pointer to an allocated bitmap object.
pPixelData	A pointer to an input buffer with raw image pixel data.
iImageSize	The size of the input image. The image is assumed to be square.

Returns

A pointer to allocated bitmap upon success, NULL upon error.

FingerLocDeleteOSBitmap

The Utility and Common API

FingerLocDeleteOSBitmap(
BITMAPINFO* pBitmap)

FingerLocDeleteOSBitmap deletes a memory buffer allocated by **FingerLocAllocOSBitmap()**.

Parameters

PARAMETER	DESCRIPTION
pBitmap	A pointer to the Device-Independent Bitmap allocated by **FingerLocAllocOSBitmap()**.

Returns

TBD.

FingerLocDeleteUser

The Utility and Common API

FingerLocDeleteUser(
tsFL_ID_INFO* pUserID)

FLRestartCommPort removes identification and fingerprint data for a given user from the FingerLoc database.

Parameters

PARAMETER	DESCRIPTION
pUserID	A pointer to a **tsFL_ID_INFO** structure specifying the user to be removed.

Returns

FL_OK	The user data was removed.
FL_NO_MATCH	The specified user does not exist in the database
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocEnroll

The Convenience API

```
FingerLocEnroll(
        tsFL_ID_INFO* pUserID
        uint16 uiFingerToEnroll
        void* pTemplatePtr
        int16 iTemplateStorageSize
        int16* piTemplateResultSize
        tsFL_MATCH_CRITERIA* pMatchSetup)
```

In the enrollment mode, **FingerLocEnroll** displays a window to guide the user through the enrollment process. This function controls the user interface, including image feedback windows, message boxes, and dialog boxes during enrollment. The function blocks (does not return to the caller) until the enrollment process is complete.

This function can also be used to obtain the size of an enrollment template by passing NULL values for the both the pointer to the enrollment user information and the pointer to the return template buffer. To receive the reference template for an enrollment, an application typically calls this function to obtain the size of a reference template, allocates memory for the template, and then calls the function again in enroll mode.

Parameters

PARAMETER	DESCRIPTION
pUserID	A pointer to an allocated **tsFL_ID_INFO** structure that specifies the user to enroll. If NULL, no enrollment data is stored in the FingerLoc database.
uiFingerToEnrollL	Specifies the number of the finger being enrolled (1 through 10).
pTemplatePtr	A pointer to an allocated buffer used to return the reference template. If NULL, no template data is returned to the calling program.
iTemplateStorageSize	The size of the allocated template buffer passed in during enrollment. Must be set to zero during template size return.
piTemplateResultSize	A pointer to a variable used to return the size of an enroll template. This value is returned when both **pUserID** and **pTemplatePtr** are NULL.
pMatchSetup	Not used.

FingerLocEnroll (continued)

Returns

FL_NO_MATCH	The finger on the sensor was enrolled, the reference template was saved to the FingerLoc database and written to the supplied buffer.
FL_MAX_USER_FINGERS_ENROLLED	User was not enrolled - insufficient memory was available.
FL_FUNCTION_FAILED	A system failure occurred - an error has resulted or could not communicate with the FingerLoc server.
FL_OUT_OF_MEMORY	The User was not enrolled - insufficient memory was available to perform the enrollment operation.
NO_FINGER_IMAGE	The User was not enrolled - a valid fingerprint was not acquired.
FINGER_NO_CORE	The User was not enrolled - could not determine a required core within the finger image.
FL_USER_CANCELED	The User canceled the operation.
Other	See the error codes in **FLStdAPI.h**.

FingerLocGetCurrentImage

The Image Capture API

```
FingerLocGetCurrentImage(
                    FL_CONTEXT_HANDLE ContextHnd,
                    CONST int16 iOutputDPI,
                    CONST BOLL bInvertImage)
```

FingerLocGetCurrentImage returns an image during Image Capture mode. This function allocates a buffer, places the image to be returned into the buffer and returns a pointer to the buffer. A NULL pointer is returned if no image is available. The application must call **FingerLocReleaseImage()** to free the image buffer, each time this function returns a non-NULL pointer.

The image is returned as a linear array of bytes, one byte per pixel, with a gray-level value in the range of 0-255. The bytes are in row-order beginning with the top row. The number of rows and columns is based on the specified image density (128, 192, or 256 DPI).

Parameters

PARAMETER	DESCRIPTION
ContectHnd	The context handle returned by **FingerLocInit()**.
iOutputDPI	Dots-per-inch (DPI) density requested for the returned image, IMAGE_250_DPI (default sensor size 128x128 pixels), IMAGE_375_DPI (192x192 pixels), IMAGE_500_DPI (256x256 pixels).
bInvertImage	If TRUE, the image data is inverted so that black pixels are output as white and white pixels are output as black.

Returns

A pointer to a buffer that holds the current image if an image is available or NULL if no image is currently available (wait, then call function again).

FingerLocGetImageState


```
FingerLocGetImageState(
                    FINGERLOC_HANDLE hContext
                    tsFL_IMAGE_STATE* psImageState)
```

FingerLocGetImageState is used to determine if the FAS has detected a finger on the sensor surface.

Parameters

PARAMETER	DESCRIPTION
hContext	This is an obsolete parameter. It is preserved for backward compatibility only. All new implementations should pass NULL for this parameter.
psImageState	A pointer to an allocated **tsFL_IMAGE_STATE** structure for the returning image state.

Returns

FL_OK	Normal return
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocGetLeds


```
FingerLocGetLeds(
                    uint32* pRtnLedState )
```

FingerLocGetLeds retrieves the state of the sensor's general-purpose outputs ("LED") bits.

Parameters

PARAMETER	DESCRIPTION
pRtnLedState	A pointer to the value to be filled upon successful return. Only the low four bits of the data are used. Definitions of the LED bits and the names are in the file **FLStdAPI.h**.

Returns

FL_OK	Normal return.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocGetNextUser


```
FingerLocGetNextUser(
                    FINGERLOC_HANDLE pContext
                    /tsFL_ID_INFO* pRtnIdInfo)
```

FingerLocGetNextUser retrieves the **tsFL_ID_INFO** structure of the next user in the FingerLoc database. If the **iUserId** member of the **tsFL_ID_INFO** structure is NULL, data is returned for the first user in the database. Otherwise, if this member is the ID of an enrolled user, this function will overwrite the data in the user information structure with that of the next user stored in the FingerLoc database.

If the specified user is the last user in the database, **FL_OK** is returned and NULL strings are written to the user information structure.

Parameters

PARAMETER	DESCRIPTION
pContext	A pointer to the context.
puiStatusInfo	A pointer to the allocated **tsFL_ID_INFO** structure used for input and output data.

Returns

FL_OK	Normal return.
FL_USER_NOT_FOUND	The specified user was not found in the database.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocGetResultDetails


```
FingerLocGetResultDetails(
                        char* pstrResultMsg
                        uint32* puiStatusInfo)
```

FingerLocGetResultDetails retrieves the last fault code, fault message string, and system information. The system status information is returned as a series of flags. These flags can be examined to determine the type of fault (fatal or error) that has occurred, in addition to what portions of the FAS are not operating correctly. The front-end application can make decisions on how to proceed based on this information.

Fault codes are found in file **FLStdApi.h** and start with **FLERR_**. System status flags are found in file **FLStdApi.h** and start with **FLSTAT_**.

Parameters

PARAMETER	DESCRIPTION
pstrResultMsg	A pointer to a char array to be filled out with result text message that describes the fault. NULL if message is not desired.
puiStatusInfo	A pointer to an allocated **uint32** for returning system status information. NULL if status information is not desired.

Returns

FL_OK or communications error code. If an error code is returned future calls to this function will probably fail.

See the error codes in **FLStdAPI.h**.

FingerLocGetVersion


```
FingerLocGetVersion(
                    int8* iHwVersion,
                    int8* iServerVersion,
                    int8* iDllVersion )
```

FingerLocGetVersion gets the version numbers of the sensor hardware, the FingerLoc server software, and the DLL software.

Parameters

PARAMETER	DESCRIPTION
iHwVersion	A pointer to an allocated byte buffer for returning an ASCII string specifying the sensor version. The maximum string size is 32 bytes.
iServerVersion	A pointer to an allocated byte buffer for returning an ASCII string specifying the server version. The maximum string size is 32 bytes.
iDllVersion	A pointer to an allocated byte buffer for returning an ASCII string specifying the DLL version. The maximum string size is 32 bytes.

Returns

FL_OK or communications error code. If an error code is returned, future calls to this function will probably fail. The application should try to release and re-acquire the communications port before attempting to retry this function (See **FLGetCommPort()** and **FLReleaseCommPort()**).

See the error codes in **FLStdAPI.h**.

FingerLocIdentify

The Convenience API

```
FingerLocIdentify(
                tsFL_ID_INFO* pUserID
                tsFL_MATCH_CRITERIA* pMatchSetup)
```

FingerLocIdentify displays an identification window to guide the user through the identification process. This function controls the User interface including image feedback windows, message boxes, and dialog boxes during the operation. The function blocks – that is, does not return to the caller - until the identification is complete.

Parameters

PARAMETER	DESCRIPTION
pUserID	A pointer to an allocated **tsFL_ID_INFO** structure for returning information specifying the identified user.
pMatchSetup	Not used.

Returns

FL_MATCH	The finger on the sensor matched the specified User.
FL_NO_MATCH	The finger on the sensor did not match the specified User.
FL_FUNCTION_FAILED	System failure - an error has resulted or could not communicate with the FingerLoc server.
FL_OUT_OF_MEMORY	Operation incomplete – there was insufficient memory to perform the Validate ID operation.
NO_FINGER_IMAGE	Unable to perform the operation - a useable fingerprint was not acquired from the sensor.
FINGER_NO_CORE	Unable to perform the operation - could not determine a core within the finger image.
FL_USER_CANCELED	The User canceled the operation.
Other	See the error codes in **FLStdAPI.h**.

FingerLocInit

The Utility and Common API

```
FingerLocInit(
        FINGERLOC_SECURITY_INFO* pSecurityInfo)
```

FingerLocInit initializes the FAS. This function is called to obtain a handle that is required as a parameter for some FingerLoc API functions. An application typically calls this function once during initialization and then passes the handle value to any functions that require it. Calling this function requires a corresponding call to **FingerLocClose()** prior to application shutdown.

Parameters

PARAMETER	DESCRIPTION
pSecurityInfo	Not used.

Returns

A handle (pointer) to a context upon success, or NULL if the initialization failed.

FingerLocOpenStream

The Image Capture API

```
FingerLocOpenStream(
                    FL_CONTEXT_HANDLE ContextHnd
                    BOOL bDisplayOn)
```

FingerLocOpenStream puts the FAS into Image Capture mode. In this mode, the FAS saves incoming sensor images into a buffer. The application retrieves the saved images by calling the **FingerLocGetCurrentImage()** function. The FAS discards any incoming images when the buffer contains a previously stored image that has not been retrieved by the application.

Parameters

PARAMETER	DESCRIPTION
ContectHnd	The context handle returned by **FingerLocInit()**
bDisplayOn	If TRUE, the FAS creates a window and displays fingerprint images.

Returns

FL_OK	The operation was successful - system has been placed into image streaming mode.
FL_STREAM_OPEN	The image steam was already open.
FL_INVALID_CONTEXT	The operation was not successful - the context handle was invalid.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocReleaseImage

The Image Capture API

```
FingerLocReleaseImage(
                    FL_CONTEXT_HANDLE ContextHnd
                    tsFL_IMAGE_STRUCT* pImage)
```

FingerLocReleaseImage deallocates the image buffer allocated by **FingerLocGetCurrentImage()**.

Parameters

PARAMETER	DESCRIPTION
ContectHnd	The context handle returned by **FingerLocInit()**.
pImage	A pointer to the image to release. This value must match the value returned by **FingerLocGetCurrentImage()**.

Returns

FL_OK	Operation was successful.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocResample150

The Utility and Common API

```
FingerLocResample150(
                    uint8* pImage
                    uint16 iImageSize
                    uint8* pNewImage)
```

FingerLocResample150 magnifies a square input image by a factor of 1.5. For example, a 128 x 128-pixel input image results in a 192 x 192-pixel image.

Parameters

PARAMETER	DESCRIPTION
pImage	A pointer to a block of bytes containing the input image formatted with one byte per pixel.
pImageSize	The size of the input image. The image is assumed to be square.
pNewImage	A pointer to an allocated output block.

Returns

FL_OK	Normal return.
FL_BAD_POINTER	Unexpected NULL pointer.

FingerLocResample200

The Utility and Common API

```
FingerLocResample200(
                    uint8* pImage
                    uint16 iImageSize
                    uint8* pNewImage)
```

FingerLocResample200 magnifies a square input image by a factor of 2. For example, a 128 x 128-pixel input image results in a 256 x 256-pixel image.

Parameters

PARAMETER	DESCRIPTION
pImage	A pointer to a block of bytes containing the input image formatted with one byte per pixel.
pImageSize	The size of the input image. The image is assumed to be square.
pNewImage	A pointer to an allocated output block.

Returns

FL_OK	Normal return.
FL_BAD_POINTER	Unexpected NULL pointer.

FingerLocSetLeds

FingerLocSetLeds(
uint32 iNewLedState)

FingerLocSetLeds toggles general-purpose outputs on and off by setting the sensor's "LED" bits.

Parameters

PARAMETER	DESCRIPTION
iNewLedState	A 32-bit integer containing the requested new LED bits. Only the low four bits of the data are used. Definitions of the LED bits and the names are in the file **FLStdAPI.h**.

Returns

FL_OK	Normal return.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FingerLocValidateFingers

The Convenience API

```
FingerLocValidateFingers(
                    void* pTemplateData
                    tsFL_MATCH_CRITERIA* pMatchSetup)
```

FingerLocValidateFingers displays a validation window to guide the user through the validation process. This function controls the user interface including image feedback windows, message boxes and dialog boxes during the operation. The function "blocks" – that is, does not return to the caller - until the validation is complete.

Parameters

PARAMETER	DESCRIPTION
pTemplateData	A pointer to a reference template to be matched against the finger placed on the sensor. The template can be either a large reference template that was returned by an Enrollment API function, or it can be a small reference template that was extracted from a large template using **FLConvertNormalReferenceTemplateToSmall()**.
pMatchSetup	Not used.

Returns

FL_MATCH	The finger on the sensor matched the specified user.
FL_NO_MATCH	The finger on the sensor did not match the specified user.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
FL_OUT_OF_MEMORY	Operation incomplete – there was insufficient memory to perform the Validate ID operation.
NO_FINGER_IMAGE	Unable to perform the operation - a valid fingerprint was not acquired from the sensor.
FINGER_NO_CORE	Unable to perform the operation - could not determine a required core within the finger image.
FL_USER_CANCELED	The User canceled the operation.
Other	See the error codes in **FLStdAPI.h**.

FingerLocValidateID

The Convenience API

```
FingerLocValidateID(
                    tsFL_ID_INFO* pUserID
                    tsFL_MATCH_CRITERIA* pMatchSetup)
```

FingerLocValidateID displays a Validate ID window to guide the user through the validation process. This function controls the user interface including image feedback windows, message boxes and dialog boxes during the operation. The function blocks (does not return to the caller) until the validation is complete.

Parameters

PARAMETER	DESCRIPTION
pUserID	A pointer to a user identification structure that specifies the user to validate. The **iUserId** field in this **tsFL_ID_INFO** structure must contain a non-NULL string.
pMatchSetup	Not used.

Returns

FL_MATCH	The finger on the sensor matched the specified user.
FL_NO_MATCH	The finger on the sensor did not match the specified user.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
FL_OUT_OF_MEMORY	Operation incomplete - there was insufficient memory to perform the Validate ID operation.
NO_FINGER_IMAGE	Unable to perform the operation - a valid fingerprint was not acquired from the sensor.
FINGER_NO_CORE	Unable to perform the operation - could not determine a required core within the finger image.
FL_USER	The User canceled the operation.
Other	See the error codes in **FLStdAPI.h**.

FLAbortTransaction

FLAbortTransaction(
FL_TRANSACTION_ID uiTransactionID)

FLAbortTransactionID provides the application with a method to terminate an open transaction.

Parameters

PARAMETER	DESCRIPTION
uiTransactionID	Transaction ID returned by the function that began the current open transaction. The *Begin Transaction* functions are: **FLBeginEnroll()**, **FLBeginVerify()**, **FLBeginIdentify()**, **FLBeginAcquireImage()**, and **FLBeginValidate()**.

Returns

TBD

FLBeginAcquireImage

The Fingerprint Services API

```
FLBeginAcquireImage(
                    FL_NOTIFY_CALLBACK pCallbackProc
                    uint32 uiCallerDWord
                    uint16 uiOperationType
                    void* pVerifyTemplate)
```

FLBeginAcquireImage is called to initiate an image Acquisition transaction. A non-zero transaction ID is returned indicating that the transaction started.

After starting a transaction, the calling program receives synchronous or asynchronous messages containing display images and state change notifications. A message of type **FL_API_ACQUIRE_DATA_RDY** indicates that the final results of the transaction are available. On receipt of this message, the application calls **FLEndAcquireImage()** to receive the image data.

Parameters

PARAMETER	DESCRIPTION
pCallbackProc	A pointer to a procedure to receive notifications of state changes during the Acquisition transaction.
uiCallerDWord	A 32-bit value to be returned as the first value in the parameter list when the FAS calls the application's Callback procedure.
uiOperationType	Not used.
pVerifyTemplate	A pointer to a template from a previous enrollment. If non-NULL, the sensor control parameters contained in the template are used to initialize the sensor to reduce the time required to acquire a good image.

Returns

A transaction number to be used by the application to track the operation of the acquire process. If NULL, the system was unable to start the transaction and a call to **FingerLocGetResultsDetails** returns the cause of the API failure. This could be caused by a transaction already being open, hardware failure, communications failure, and so on.

FLBeginEnroll

The High Performance API

```
FLBeginEnroll(
        FL_NOTIFY_CALLBACK pCallbackProc
        uint32 uiCallerDWord
        tsFL_MATCH_CRITERIA* pMatchSetup
        tsFL_ID_INFO* pUserID
        FL_FINGER_CODE iWhichFinger
        uint16* puiMaxTemplateSize)
```

FLBeginEnroll begins an Enrollment transaction. A non-zero transaction number is returned to indicate that the transaction started.

After starting a transaction, the calling program receives synchronous or asynchronous messages containing display images and state change notifications. A message of type **FL_API_ENROLL_DATA_RDY** indicates that the final results of the transaction are available, Upon receipt of this message, the application calls **FLEndEnroll()** to receive the results of the enrollment and to end the transaction.

Parameters

PARAMETER	DESCRIPTION
pCallbackProc	A pointer to the Callback procedure to receive asynchronous display images and transaction state changes during the transaction.
uiCallerDWord	A 32 bit value to be returned as the first value in the parameter list when the FAS calls the application's Callback procedure.
pMatchSetup	Not used.
pUserID	A pointer to a **tsFL_ID_INFO** structure specifying the user to enroll If NULL, enroll data is not saved to the FingerLoc database.
iWhichFinger	The number of the finger to enroll.
puiMaxTemplateSize	A pointer to returned size of an Enrollment template

FLBeginEnroll (continued)

Returns

An **FL_TRANSACTION_ID** used by the application to track the status of the transaction. If NULL, the system was unable to begin the transaction and a call to **FingerLocGetResultsDetails()** returns the cause of the API failure - a transaction already open, a hardware failure, a communications failure, and so on.

FLBeginIdentify

The High Performance API

```
FLBeginIdentify(
        FL_NOTIFY_CALLBACK pCallbackProc
        uint32 uiCallerDWord
        tsFL_MATCH_CRITERIA* pMatchSetup)
```

FLBeginIdentify initiates an identification transaction. A non-zero transaction number is returned, indicating that the transaction started.

After starting a transaction, the calling program receives synchronous or asynchronous messages containing display images and state change notifications. A message of type **FL_API_IDENTIFY_DATA_RDY** indicates the final results of the transaction are available.

Upon receipt of this message, the application calls **FLEndIdentify()** to receive the results of the validation and to end the transaction.

Parameters

PARAMETER	DESCRIPTION
pCallbackProc	A pointer to the callback procedure to receive asynchronous display images and transaction state changes during the transaction.
uiCallerDWord	A 32 bit value to be returned as the first value in the parameter list when the FAS calls the application's callback procedure.
pMatchSetup	A pointer to structure containing the match requirements for this operation. If the pointer is NULL, then the default requirements are used.

Returns

A transaction number to be used by the application to track the status of the transaction. If NULL, the system was unable to begin the transaction and a call to **FingerLocGetResultsDetails()** returns the cause of the API failure - a transaction already open, a hardware failure, a communications failure, and so on.

FLBeginValidateID

The High Performance API

```
FLBeginValidateID(
                    FL_NOTIFY_CALLBACK pCallbackProc
                    uint32 uiCallerDWord
                    tsFL_ID_INFO* pUserID
                    tsFL_MATCH_CRITERIA* pMatchSetup)
```

FLBeginValidateID initiates a “Validate ID” transaction. A non-zero transaction number is returned, indicating that the transaction started.

After starting a transaction, the calling program receives synchronous or asynchronous messages containing display images and state change notifications. A message of type **FL_API_VALIDATE_DATA_RDY** indicates the final results of the transaction are available, Upon receipt of this message, the application calls **FLEndValidateID()** to receive the results of the validation and to end the transaction.

Parameters

PARAMETER	DESCRIPTION
pCallbackProc	A pointer to the callback procedure to receive asynchronous display images and transaction state changes during the transaction.
uiCallerDWord	A 32 bit value to be returned as the first value in the parameter list when the FAS calls the application's callback procedure.
pUserID	A pointer to a **tsFL_ID_INFO** structure that specifies the user to validate.
pMatchSetup	A pointer to structure containing the match requirements for this operation. If NULL, default requirements are used.

Returns

A transaction number used by the application to track the status of the transaction. If NULL, the system was unable to begin the transaction and a call to **FingerLocGetResultsDetails()** returns the cause of the API failure - a transaction already open, a hardware failure, a communications failure, and so on.

FLBeginVerify

The High Performance API

```
FLBeginVerify(
        FL_NOTIFY_CALLBACK pCallbackProc
        uint32 uiCallerDWord
        void* pTemplatePtrs
        int16 iNumberTemplates
        tsFL_MATCH_CRITERIA* pMatchSetup)
```

FLBeginVerify initiates a verification transaction. A non-zero transaction number is returned to indicate that the transaction started.

After starting a transaction, the calling program receives synchronous or asynchronous messages containing display images and state change notifications. A message of type **FL_API_VERIFY_DATA_RDY** indicates that the final results of the transaction are available,

Upon receipt of this message, the application calls **FLEndVerify()** to receive the results of the verification and to end the transaction.

Parameters

PARAMETER	DESCRIPTION
pCallbackProc	A pointer to the callback procedure to receive asynchronous display images and transaction state changes during the transaction.
uiCallerDWord	A 32 bit value to be returned as the first value in the parameter list when the FAS calls the application's callback procedure.
pTemplatePtrs	A pointer to an array of template pointers. The user will be matched against these templates.
iNumberTemplates	The number of pointers in the array.
pMatchSetup	A pointer to a structure containing the match requirements for this operation. If NULL, default requirements are used.

FLBeginVerify (continued)

Returns

An **FL_TRANSACTION_ID** used by the application to track the status of the transaction. If NULL, the system was unable to begin the transaction and a call to **FingerLocGetResultsDetails()** returns the cause of the API failure - a transaction already open, a hardware failure, a communications failure, and so on.

FLBuildMatchTemplate

The Fingerprint Services API

```
FLBuildMatchTemplate(
                    uint8* pRawImage,
                    void* pMatchTemplateStorage)
```

Given an image as input, **FLBuildMatchTemplate** extracts a match template. This function is typically called before calling the function **FLMatchTemplatePair()** which requires a match template as input.

Parameters

PARAMETER	DESCRIPTION
pRawImage	A pointer to a raw image. The **FLBeginAcquireImage()** and **FLEndAcquireImage()** functions are used to obtain a raw image.
pMatchTemplateStorage	A pointer to an allocated match template buffer. Use **FLGetMatchTemplateSize()** to size this buffer.

Returns

FL_OK	Successful error return.
FL_BAD_POINTER	Returned if either input parameter is NULL.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.

FLBuildOSBmpHeader

The Fingerprint Services API

```
FLBuildOSBmpHeader(
                    uint8* pRawImage
                    void* pOSBmpHeader)
```

FLBuildOSBmpHeader fills out an allocated bitmap header of size **FLGetOSBmpHeaderSize()** with values and pixel data if required. The result is used to display the image in the current operating system.

Parameters

PARAMETER	DESCRIPTION
pRawImage	A pointer to raw sensor image data.
pOSBmpHeader	A pointer to an allocated OS-dependent bitmap information structure. If the OS requires a local copy of the pixels in a bitmap object, this function is used to move pixel data into the bitmap information structure. The format of the bitmap structure is specific to the operating system and is described in the API documentation for each platform. If the pointer to the image data is NULL, or if the OS bitmap information does not contain pixel data, image data is not copied. In the case of Windows, **pOSBmpHeader** points to a **BITMAPINFO** structure containing a **BITMAPINFOHEADER** followed by a color palette, which is an **RGBQUAD** array. A **BITMAPINFO** structure contains no pixel data.

Returns

FL_OK	Normal return.
FL_BAD_POINTER	Unexpected NULL pointer.

FLBuildReferenceTemplate

The Fingerprint Services API

FLBuildReferenceTemplate(
uint8* pRawImage,
void* pReferenceTemplate)

Given an input image, **FLBuildReferenceTemplate** builds a reference or enroll type of template. This function is typically called before calling the function **FLMatchTemplatePair()** which requires a reference template as input.

Parameters

PARAMETER	DESCRIPTION
pRawImage	A pointer to a raw image. The **FLBeginAcquireImage()** and **FLEndAcquireImage()** functions are used to obtain a raw image.
pReferenceTemplate	A pointer to an allocated buffer to receive the reference template. Use **FLGetReferenceTemplateSize()** to size this buffer.

Returns

FL_OK	Successful error return
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.

FLContinueTransaction

FLContinueTransaction(
FL_TRANSACTION_ID uiTransactionID)

FLContinueTransaction returns a pointer to the next API event message to be processed for the current transaction.

This function provides a method for synchronous event processing by allowing the calling application to poll the FAS for event messages during a transaction. The polling method to obtain API event messages can be used when it is not practical for the application to support the asynchronous callback method for handling API event messages.

The **FLReleaseMessage()** function must be called to release the current message and free the returned message buffer once the application has processed the API event message.

Parameters

PARAMETER	DESCRIPTION
uiTransactionID	The Transaction ID returned by the function that began the current open transaction. The Begin Transaction functions are: **FLBeginEnroll()**, **FLBeginVerify()**, **FLBeginIdentify()**, **FLBeginAcquireImage()**, and **FLBeginValidate()**.

Returns

The function returns a pointer to a message if an event message is available or NULL if no message is available. In the latter case, the application typically sleeps for a short period (10 - 100 milliseconds) and calls the function again. NULL is also returned in the event of an error or if an invalid transaction ID is specified.

FLConvertNormalReferenceTemplateToSmall

The Fingerprint Services API

FLConvertNormalReferenceTemplateToSmall(
void* pRtnSmallTmpltBffr,
void* pInputNormalTmplt)

FLConvertNormalReferenceTemplateToSmall extracts a small reference template from a normal (large) template.

Parameters

PARAMETER	DESCRIPTION
pRtnSmallTmplBffr	A pointer to an allocated buffer for returning a small template. Use **FLGetSmallReferenceTemplateSize()** to allocate this buffer.
pInputNormalTmplt	A pointer to the normal, or large, template to be converted.

Returns

FL_OK	Normal return
FL_BAD_POINTER	Unexpected NULL pointer

FLEndAcquireImage

The Fingerprint Services API

```
FLEndAcquireImage(
                    FL_TRANSACTION_ID uiCurrentID
                    uint8* pRawImage)
```

FLEndAcquireImage ends an image acquisition transaction and returns the result.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The transaction ID provided by the transaction "begin" call.
pRawImage	A pointer to an allocated memory buffer used to return the final image. Use **FLGetImageBufferSize()** to size this buffer. On a successful return, the data in this buffer begins with pixel data and may contain other support data following the pixel data. This allows the caller to easily access the pixel data if needed for display, extraction or other use.

Returns

FL_OK	Normal return value.
FL_TIMEOUT	Unable to acquire a valid image - the application should not see the data in the return image buffer.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndEnroll

The High Performance API

```
FLEndEnroll(
        FL_TRANSACTION_ID uiCurrentID
        void* pTemplatePtr
        int16* piTemplateResultSize
        FINGERLOC_ID_INFO* pUserID)
```

FLEndEnroll returns the results of the enrollment and terminates the enrollment transaction.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID provided by the **FLBeginEnroll** function.
pTemplatePtr	A pointer to an allocated buffer for returning the reference template. Use **FLGetReferenceTemplateSize()** to size this buffer. If NULL, no template is returned.
piTemplateResultSize	A pointer to the number of bytes written to the template buffer.
pUserID	Pointer to a structure specifying the user to enroll. If NULL, enrollment data is not saved in the FingerLoc database.

Returns

FL_OK	The enrollment was successful.
FL_NO_IMAGE	No image was acquired
FL_BAD_IMAGE	No suitable image was acquired
FL_FUNCTION_FAILED	Unable to save template to database
Other	See the error codes in **FLStdAPI.h**.

FLEndIdentify

The High Performance API

```
FLEndIdentify(
        FL_TRANSACTION_ID uiCurrentID
        tsFL_ID_INFO* pUserID)
```

FLEndIdentify terminates an identification transaction and returns the result of the transaction.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID provided by the **FLBeginIdentify** call.
pUserID	A pointer to a User ID structure or NULL. If non-NULL and a match occurs, this structure is filled in with the matched user ID information.

Returns

FL_MATCH	A match was found in the database
FL_NO_MATCH	No match was found in the database
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndIdentifyEx

The High Performance API

```
FLEndIdentifyEx(
        FL_TRANSACTION_ID uiCurrentID,
        tsFL_ID_INFO* pUserID,
        tsFL_MATCH_RESULTS* pMatchResults,
        void* pParamStruct );
```

FLEndIdentifyEx terminates an identification transaction and returns the result of the transaction. This function is the extended version of **FLEndIdentify()**, with an additional parameter used to return match results indicating the strength of the acceptance or rejection.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID provided by the **FLBeginIdentify** call.
pUserID	A pointer to a User ID structure or NULL. If non-NULL and a match occurs, this structure is filled in with the matched user ID information.
pMatchResults	If non-NULL, returns confidence information about the match result.
pParamStruct	Reserved for future use.

Returns

FL_MATCH	The user ID was validated
FL_NO_MATCH	The user ID was not validated
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndValidateID

The High Performance API

```
FLEndValidateID(
                FL_TRANSACTION_ID uiCurrentID )
```

FLEndValidateID terminates a validation transaction and returns the result of the transaction.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID returned by the **FLBeginValidateID** call.

Returns

FL_MATCH	The user ID was validated
FL_NO_MATCH	The user ID was not validated
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndValidateIDEx

The High Performance API

```
FLEndValidateIDEx(
                    FL_TRANSACTION_ID uiCurrentID,
                    tsFL_MATCH_RESULTS* pMatchResults,
                    void* pParamStruct)
```

FLEndValidateIDEx terminates a validation transaction and returns the result of the validation. This function is the extended version of **FLEndValidateID()** with an additional parameter used to return match results indicating the strength of the FAR/FRR.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID returned by the **FLBeginValidateID** call.
pMatchResults	Allocated **tsFL_MATCH_RESULTS** structure for returning confidence information about the match result.
pParamStruct	Reserved for future use.

Returns

FL_MATCH	The user ID was validated
FL_NO_MATCH	The user ID was not validated
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndVerify

The High Performance API

```
FLEndVerify(
        FL_TRANSACTION_ID uiCurrentID
        uint16* pMatchIndex)
```

FLEndVerify terminates a verification transaction and returns the result of the transaction.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID provided by the **FLBeginVerify** call.
pMatchIndex	A pointer to a variable which is set on a successful verification to the index of the template that matched the User.

Returns

FL_MATCH	A match was found
FL_NO_MATCH	No match was found
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLEndVerifyEx

The High Performance API

```
FLEndVerifyEx(
        FL_TRANSACTION_ID uiCurrentID,
        uint16* pMatchIndex,
        tsFL_MATCH_RESULTS* pMatchResults,
        void* pParamStruct )
```

FLEndVerifyEx terminates a verification transaction and returns the result of the transaction. This is the extended version of **FLEndVerify()** with an additional parameter used to return match results indicating the strength of the acceptance or rejection.

Parameters

PARAMETER	DESCRIPTION
uiCurrentID	The Transaction ID provided by the **FLBeginVerify** call.
pMatchIndex	A pointer to a variable which is set on a successful verification to the index of the template that matched the User.
pMatchResults	Returns confidence information about the match result.
pParamStruct	Reserved for future use.

Returns

FL_MATCH	The user ID was validated
FL_NO_MATCH	The user ID was not validated
FL_BAD_POINTER	Unexpected NULL pointer
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLGetCommPort


```
FLGetCommPort(
        uint8* pMuxControl
        uint16 uiCmdLen)
```

FLGetCommPort gets control of the serial port. This allows the application to inform the control software that the hardware is about to disconnect the sensor from the input port. This function could be useful when a User wishes to operate two devices, such as a sensor and a smart card reader, on the same communications port.

Parameters

PARAMETER	DESCRIPTION
pMuxControl	Byte array with data to switch the mux, as issued by AuthenTec.
uiCmdLen	The number of bytes in the array. Length of switch sequence.

Returns

FL_OK	Normal return.
FL_FUNCTION_FAILED	This is a general error return - use **FingerLocGetResultsDetails()** to receive additional details about the error.
Other	See the error codes in **FLStdAPI.h**.

FLGetImageBufferSize

FLGetImageBufferSize()

FLGetImageBufferSize gets the size of the image buffer block required to store fingerprint image data. This function is typically called to allocate a memory buffer prior to calling a function that returns image data.

Parameters

PARAMETER	DESCRIPTION
none	There are no parameters associated with this function.

Returns

The size of a FingerLoc image buffer. A FingerLoc image buffer begins with 128 x 128 bytes of image data followed by additional support data.

FLGetMatchTemplateSize

FLGetMatchTemplateSize()

FLGetMatchTemplateSize is used to get the size in bytes of a match template. This function is typically called prior to allocating memory for a match template.

Parameters

PARAMETER	DESCRIPTION
none	There are no parameters associated with this function.

Returns

Size in bytes of a match template.

FLGetOSBmpHeaderSize

FLGetOSBmpHeaderSize()

FLGetOSBmpHeaderSize gets the size of a bitmap header structure, specific to the current operating system, describing a displayable fingerprint image.

The size includes all components necessary for the current operating system. This can include header info, color look-up tables and pixel data if the general operating system specific method of handling displayable images (image function calls) requires bundled components. A bitmap header object can be allocated and subsequently passed to **FLBuildOSBmpHeader()** to be filled out.

Windows BMP operating system calls normally have parameters that specify both a pointer to bitmap header structure that describes the image format, and a separate pointer to the actual pixel data. A call to this function will therefore return the size required to store a bitmap header and a color look-up array when called within the Windows operating system.

Parameters

PARAMETER	DESCRIPTION
none	There are no parameters associated with this function.

Returns

The size in bytes of the bitmap header.

FLGetReferenceTemplateSize()

//

FLGetReferenceTemplateSize returns the size in bytes of a normal (“large”) reference template created by an *Enrollment* operation. This function is typically called prior to allocating memory for a reference template.

Parameters

PARAMETER	DESCRIPTION
none	There are no parameters associated with this function.

Returns

The size in bytes of a large reference template.

FLGetSmallReferenceTemplateSize

The Fingerprint Services API

FLGetSmallReferenceTemplateSize()

FLGetSmallReferenceTemplateSize returns the size in bytes of a small reference template. Typically, this function is called prior to allocating memory for a small reference template. See information below regarding reduced sized templates.

Reduced sized templates are useful in limited storage space applications. Reduced sized templates are created by calling **FLConvertNormalReferenceTemplateToSmall()**. Large templates are always exported by FingerLoc Enrollment functions. An application can extract a small template from a large one and store only the reduced sized template. A reduced size template can be passed to the FingerLoc Matching API functions.

Larger templates are required for high speed, one-to-many identification operations. Reduced-sized templates can be used in one-to-many matching applications. However, for large databases, their use can result in significantly reduced performance and are not recommended.

Parameters

PARAMETER	DESCRIPTION
none	There are no parameters associated with this function.

Returns

Size in bytes of a small reference template.

FLMatchTemplatePair

The Fingerprint Services API

```
FLMatchTemplatePair(
                    void* pReferenceTemplate
                    void* pMatchTemplate)
```

FLMatchTemplatePair compares a Reference (or Enrollment) template to a Matching template to determine if the templates represent the same fingerprint. Default match criteria are used to determine if the templates match.

This is similar to the extended version of this function, **FLMatchTemplatePairEx()**, which has added parameters to specify the match criteria and return the achieved match results.

Parameters

PARAMETER	DESCRIPTION
pReferenceTemplate	A pointer to a reference template. A reference template is returned by the Enrollment API functions or by **FLBuildReferenceTemplate()**.
pMatchTemplate	A pointer to a match template. A match template is returned by the **FLBuildMatchTemplate()** function.

Returns

FL_MATCH	The templates represent the same finger
FL_NO_MATCH	The templates are not for the same finger
FL_BAD_POINTER	A template pointer was NULL
FL_FUNCTION_FAILED	This is a general error return - a call to **FingerLocGetResultsDetails** returns additional details about the result.

FLMatchTemplatePairEx


```
FLMatchTemplatePairEx(
                    void* pReferenceTemplate
                    void* pMatchTemplate,
                    tsFL_MATCH_CRITERIA* pMatchCriteria,
                    tsFL_MATCH_RESULTS* pMatchResults,
                    void* pParamStruct)
```

FLMatchTemplatePairEx compares a Reference (or Enrollment) template to a Matching template to determine if the templates represent the same fingerprint.

This call is similar to the function **FLMatchTemplatePair()**, but has additional parameters to specify match criteria and to receive match results that indicate the strength of the match.

Parameters

PARAMETER	DESCRIPTION
pReferenceTemplate	A pointer to a reference template. A reference template is returned by the enroll API functions and by the **FLBuildReferenceTemplate()**.
pMatchTemplate	A pointer to a match template. A match template is returned by **FLBuildMatchTemplate().**
pMatchCriteria	A pointer to match criteria structure that specifies the FAR (False Acquisition Rate) and FRR (False Rejection Rate) criteria. If NULL, default criteria are used.
pMatchResults	A pointer to an allocated **tsFL_MATCH_RESULTS** structure for returning match results in terms of FAR and FRR achieved. If NULL, no results are returned.
pParamStruct	Reserved for future use.

FLMatchTemplatePairEx (continued)

Returns

FL_MATCH	The templates are for the same finger
FL_NO_MATCH	The templates are not for the same finger
FL_BAD_POINTER	Unexpected NULL template pointer.
FL_FUNCTION_FAILED	General error return – use **FingerLocGetResultsDetails()** to receive additional details about the error.

FLReadCommPort


```
FLReadCommPort(
                uint8* plOData
                uint16 uilObufSize
                uint16* uilOLen
                uint32 uiMaxReadTime)
```

FLReadCommPort reads a byte stream from the communications port. The application must have successfully called **FLGetCommPort** before using this function.

Parameters

PARAMETER	DESCRIPTION
plOData	A pointer to data to be sent.
uilObufSize	The number of bytes in the buffer.
uiReadCount	A pointer to a variable to receive the transfer count.
uiMaxReadTime	The maximum number of milliseconds to wait for data of **uilObufSize**. If zero, the wait time is calculated based on the current baud rate, and the number of bytes to read. Usually the application will set this to zero and allow the system to decide about timeouts.

Returns

FL_OK or communications error code. If an error code is returned, future calls to this function will probably fail. The application should try to release and re-acquire the communications port before attempting to retry this function (See **FLGetCommPort()** and **FLReleaseCommPort()**).

See the error codes in **FLStdAPI.h**.

FLReleaseCommPort

FLReleaseCommPort(
uint8* pMuxControl
uint16 uiCmdLen)

FLReleaseCommPort releases control of the serial communications port

Parameters

PARAMETER	DESCRIPTION
pMuxControl	Byte array with data to switch the mux as issued by AuthenTec.
uiCmdLen	The number of bytes in the array. Length of switch sequence.

Returns

FL_OK or communications error code. If an error code is returned, future calls to this function will probably fail. The application should try to release and re-acquire the communications port before attempting to retry this function (See **FLGetCommPort()** and **FLReleaseCommPort()**).

See the error codes in **FLStdAPI.h**.

FLReleaseMessage

The Fingerprint Services API

```
FLReleaseMessage(
                    tsFL_API_MSG* pTransactionMsg )
```

FLReleaseMessage releases an API event message received by an application's callback function when the application is receiving API event messages in asynchronous mode or an API event message received by calling **FLContinueTransaction()** when the application is using polled mode.

Parameters

PARAMETER	DESCRIPTION
pTransactionMsg	A pointer to the API event message provided by **FLContinueTransaction** function or the API event-message pointer received by the asynchronous callback function. This parameter may not be NULL.

Returns

TBD

FLWriteCommPort

The Utility and Common API

```
FLWriteCommPort(
                    uint8* pIOData
                    uint16 uiIOLen)
```

FLWriteCommPort writes a byte stream to the communications port. The application must have successfully called **FLGetCommPort** before using this function.

Parameters

PARAMETER	DESCRIPTION
pIOData	A pointer to data to be sent.
uiIOLen	The number of bytes in the buffer to send.

Returns

FL_OK or communications error code. If an error code is returned, future calls to this function will probably fail. The application should try to release and re-acquire the communications port before attempting to retry this function (See **FLGetCommPort()** and **FLReleaseCommPort()**).

See the error codes in **FLStdAPI.h**.

AuthenTec, Inc.
Post Office Box 2719
Melbourne, Florida 32902-2719
321-308-1300
www.authentec.com
apps@authentec.com