Kofax FraudOne

Common API Specifications for GIA Engines

Version: 4.5.0

Date: 2020-11-04

Table of Contents

Preface

APIA (Automatic Payment Image Analysis) refers to the usage of algorithms, or neural networks, to extract information from images of payment transaction documents. An APIA engine is any software component that provides the capability to analyze a transaction document image. The FraudOne product utilizes image information to allow fully automated processing of large volumes of check images. By correlating APIA results with other fraud detection technologies, a significant reduction in fraud loss is achieved. The FraudOne platform is based on an open architecture that enables different APIA engine vendors to be integrated.

GIA (General Image Analysis) refers to the integration of unlimited varieties of third-party engines from the vast family of image processing engines that show benefits for image based payment processing with FraudOne. GIA provides a standardized plug-in place that enables efficient maintenance and cost effective usage of such engines for Kofax's customers.

The image processing engines targeted with this API could refer to various needs in e.g. image quality handling, in fraud detection, or in workflow automation that can be better satisfied by the integrated rather than the external employment of such features in the items processing workflow of banks.

GIA refers this broader approach of the current FraudOne release that has been derived from the earlier experiences made with the original APIA approach that was focusing on image analysis for fraud detection purposes only. Market observation and Customer feed back has proven the herewith supplied benefit for bank's item processing operations. The APIA naming convention inside the API of this earlier approach will be continued for the GIA-API as it moves on to an even broader scope in the future.

GIA Philosophy: The efficiency and effectiveness of a specific FraudOne deployment in a specific market environment is not dependent on the quality of just one single image analysis engine but rather on the right combination of the right engines in the right environment. Different customers have different needs based on the character of their fraud exposure, their legal environment, or their process boundary conditions. Kofax is committed to offering our customers varying combinations of engine types supplied by different engine vendors based on their specific needs. This GIA-API is designed to allow different vendors' engines to be integrated into a single platform in a way best fitting the needs of our customers.

Vendors wishing to integrate into Kofax's architecture are not required to implement all features of the API. It is at the vendor's discretion to decide whether they would like to specialize on specific GIA features (best of breed), or to provide the largest range features within one single engine.

This document describes Kofax's interface for integrating software engines that implement GIA features into the FraudOne platform.

Related documentation

The full documentation set for Kofax FraudOne is available at the following location:

https://docshield.kofax.com/Portal/Products/FO/4.5.0-th2k87ey6r/FO.htm

In addition to this guide, the documentation set includes the following items:

Guides

- *Kofax FraudOne Administrator's Guide*
- *Kofax FraudOne Data Warehouse Installation and Operation Guide*
- *Kofax FraudOne Extended Reporting Features and Statistics*
- *Kofax FraudOne Feature Codes*
- *Kofax FraudOne Installation and Migration Guide*
- *Kofax FraudOne Java Client Customization Guide*
- *Kofax FraudOne Java Client Customization Layer*
- *Kofax FraudOne License Management*
- *Kofax FraudOne Report Component Installation Guide*
- *Kofax FraudOne SignCheck Result Codes*
- *Kofax FraudOne Standard Reporting Features and Statistic*
- *Kofax FraudOne The Book on CRS*
- *Kofax FraudOne Thin Client Customization Guide*
- *Kofax FraudOne Thin Client Customization Layer*

Interfaces

- *Kofax FraudOne Archive Interface Server*
- *Kofax FraudOne ASV Blackbox*
- *Kofax FraudOne Global Fraud Signature Web Service Developer's Guide*
- *Kofax FraudOne Service Program Interfaces*
- *Kofax FraudOne User Login Procedure*
- *Kofax FraudOne Standard Teller Interface*
- *Kofax FraudOne Variant Cleanup Utility*

Online Help

- *Kofax FraudOne Administration Client Help*
- *Kofax FraudOne Java Client Help*
- *Kofax FraudOne Server Monitor Help*
- *Kofax FraudOne Thin Client Help*

Training

Kofax offers both classroom and computer-based training that will help you make the most of your Kofax FraudOne solution. Visit the Kofax website at www.kofax.com for complete details about the available training options and schedules.

Getting help with Kofax products

The Kofax Knowledge Base repository contains articles that are updated on a regular basis to keep you informed about Kofax products. We encourage you to use the Knowledge Base to obtain answers to your product questions.

To access the Kofax Knowledge Base, go to the Kofax website and select **Support** on the home page.

Note The Kofax Knowledge Base is optimized for use with Google Chrome, Mozilla Firefox or Microsoft Edge.

The Kofax Knowledge Base provides:

- Powerful search capabilities to help you quickly locate the information you need.

  Type your search terms or phrase into the **Search** box, and then click the search icon.
- Product information, configuration details and documentation, including release news.

  Scroll through the Kofax Knowledge Base home page to locate a product family. Then click a product family name to view a list of related articles. Please note that some product families require a valid Kofax Portal login to view related articles.
- Access to the Kofax Customer Portal (for eligible customers).

  Click the **Customer Support** link at the top of the page, and then click **Log in to the Customer Portal**.
- Access to the Kofax Partner Portal (for eligible partners).

  Click the **Partner Support** link at the top of the page, and then click **Log in to the Partner Portal**.
- Access to Kofax support commitments, lifecycle policies, electronic fulfillment details, and self-service tools.

  Scroll to the **General Support** section, click **Support Details**, and then select the appropriate tab.

Chapter 1

Common API interface for GIA Engines

General utilization

Based on past experience and future planning of Kofax's large-bank customers there will be several different areas for GIA feature utilization. The most common is the 'batch-oriented' processing for back office purposes where GIA features will be integrated into the FraudOne application server and load balancing architecture. The second most likely will be the 'online-oriented' processing of single requests coming from a FraudOne Java Client environment.

The below described common API is meant to turn integration efforts into close-to-plug-and-play efforts. This also allows easier version maintenance of engines for the vendor as well easier testing and distribution to customer installations for Kofax. In order to provide such a common API plus the necessary flexibility for features this API definition has the character of an integration framework.

The following API description defines one single common interface for all possible GIA features provided through external engines. The vendor may chose to provide a single feature or a multiple engine API. In all cases the calling layer of these engines need to follow the below defined description.

Basics

The Vendor will provide a C-API for the engine that allows running multiple thread-safe instances per process. A Kofax GIA verification client might utilize several instances of the same engine. There may be several instances of a GIA verification client running on a single machine. Per instance there will be one separate workspace under the responsibility of the engine.

One instance has a life cycle from GIA-client start through unlimited number of verification, validation or detection cycles to GIA-client stop. Initialization elements that are independent on reference image and verification item setups shall only be performed throughout initial start-up in order to keep processing overhead low.

For the time being the FraudOne environment will only provide black and white TIFF images with compression CCITT Group4 or no compression to the GIA engine. All images within those definition boundaries can be handled through this API. This includes front and back of the check as well as snippets of the check image if necessary.

Due to the typical constrains of large installations within typical lights-out environments of server farms Kofax requests that engines do not require Windows registry entries, OCX registrations, WIN-INI entries, separate installation routines or any kind of hardware (e.g. dongles) for license maintenance purposes. Engines must be deployable as part of FraudOne file packages.

Initialization and setup

Most engines need certain configuration that defines the modus operandi, which might come along as a set of configuration files or similar elements. Configuration files, if exist, need to be inside of a logical black box together with the engine's DLL so that they can be deployed along with it.

For every engine there might be a set of necessary environmental setups that are variable to the installation or the processing mode. In a stand-alone situation the vendor may opt for a local binary parameter file or a WIN-INI type configuration file or a windows registry entry. The method of deployment within the fully integrated FraudOne suite requires that such setup parameters must be integrated into the FraudOne system settings environment. Therefore the API has to provide external initialization, setup and configuration parameters that can be transmitted to their engines through API calls by the calling GIA-client. All parameters should be stored by the FraudOne system and handed over to the engine as ASCII name and value pairs. The vendor has to provide specifications for such parameters so that they can be correctly integrated into the FraudOne system configuration settings.

There are three different kinds of possible initialization and setup data:

Data to initialize an engine (DLL) on GIA start-up. The configuration settings are a part of the vendor's 'black box'. FraudOne will not hand over configuration parameters for initialization; they must be a part of the engines own environment. Optional parameters to be handed over in the initialization call are possible though.

Data to initialize a sub-engine or feature within an engine (DLL). There are reasons for process dependent configuration. Such configuration needs to be transparent to Kofax or even the bank utilizing the system in order to make business driven adjustments. These configuration values will reside within the FraudOne configuration layers and handed over to the engine with the setup call. It must be possible to re-setup such configurations during life cycle.

Data to initialize a feature/sub-engine before verification execution. There are reasons for process dependent setup parameters that can change between verification cycles. These configuration values will reside within the FraudOne system in the applicable data or control layers and handed over to the engine before a verification, validation or detection step.

Once an initialization or setup was performed, the corresponding configuration values must stay set within the workspace until overridden or cleared in order to keep initialization overheads low. Therefore engine or verification setup parameters will be set, overwritten or unset through corresponding calls and otherwise be static within the workspace. The vendor also provides a possibility to reset a feature/sub-engine parameter value to its default value.

Memory handling

In most of the cases memory allocation will be done by the calling application. The memory for Images, reference data, parameters, etc. will be allocated before calling the respective function call. Only result set allocation will be done by the engine. If the memory handed over by pointer to the engine is not sufficient to fulfil the task the engine has to return the APIA_INSUFFICIENT_MEM error code. This enables the calling application to retry the function after allocating the appropriate amount of memory.

The time when the memory will be released can differ depending on the API function call. For further details regarding memory allocation and how long this memory is valid please refer to the detailed function call specification.

Function return value

If not described otherwise all function calls return APIA_OK if successful or an error code as defined below. The calling system can call APIA_GetErrorText for further error details. Result values must be provided by the engine through the parameter list.

Analysis setup principles

Corresponding with the modus operandi of FraudOne item processing the analysis has two preparation steps.

The first step is always the handover of the item to-be-verified into the engines workspace. (This could be the items image(s) or its data or both.)

The second step is related to the nature of the task. If the verification is dependent to account profile references (check stock samples or any kind of parameter), then the second step will be the transfer of such references or reference parameters to the engines workspace.

Basically, the modus operandi requests to running references against the item, rather then items against a reference. (However, this is only a philosophical point!)

Based on the circumstances given in a specific project environment, the calling application needs the flexibility to either request a single result for a one-to-one verification or to hand over multiple references and retrieve a corresponding array of results.

Reference and item data

There are many different flavors of potential verification engines. Some verification types request only the item’s image and some need reference images and/or data. Some engines provide an independent set of logically connected calls. So a specific call could only invoke MICR information and dependent on the validation outcome a second call with full blown reference images and data will follow. In order to keep all possibilities open, while minimizing the complexity of the interface, there are some necessary definitions.

Item side information such as the full item image OR any kind of snippet from the item image OR the MICR information OR additional form-type information is considered 'item information' in the following description.

All non-item information, whether account-related OR non account-related, is considered 'reference information'. Examples of account related reference information are check-stock references, positive payees from the account history, account type information or positive-pay etc. Examples of non-account related reference information are market-type check-stocks (e.g. money orders), black-listed payee names or bank-logo images etc.

Common definitions

These are the common definitions that the vendor's DLL has to implement. They will be provided through a Kofax side c-Header file. It is possible that future versions of this document and corresponding header-file will define additional elements to be implemented by the engine.

API logging levels

These are the common definitions of the logging level(s) used within the FraudOne system. It is requested that an engine provides logging information to the calling application using a call-back procedure. The logging levels are cumulative in a way that higher levels also include all of the logging information of lower levels, e.g. the APIA_LOG_TRACE level includes Info, Warning and Error levels).

Log definition	Value	Description
APIA_LOG_OFF	0	No trace file
APIA_LOG_CRITICAL	1	Log critical errors in trace file
APIA_LOG_ERROR	2	Log critical errors and errors
APIA_LOG_INFO	3	Log critical errors, errors and infos
APIA_LOG_TRACE	4	Log full trace (includes critical errors, errors and infos)

Engines are expected not to throw their own exceptions into the Windows environment. The escalation of an error situation to the application exception logs remains a FraudOne domain.

API error codes

The API is designed in a way that each call will return a result code indicating the outcome of the action. A return value of zero always means that the procedure has finished work properly while a value below zero signals any kind of error listed below.

Engines may pass additional return codes to the calling application. These return codes can be any positive value or negative values starting from -101. Please note that the range up to -100 is reserved for the calling application only.

Additional return values provided by the call will passed via the parameter list.

Error code	Value	Description
APIA_OK	0	
APIA_NO_ITEM_IMAGE	1	No item image available
APIA_NO_REF_IMAGE	32	No reference check image available
APIA_FORM_NOT_SUPPORT	44	Form type not supported
APIA_BAD_ITEM_IMAGE	50	Item image corrupt
APIA_TWO_ITEMS	51	Already one item loaded
APIA_NO_ITEM_DATA	52	No item data available
APIA_DATA_INCOMPLETE	53	Item data incomplete
APIA_NO_REF_DATA	54	No reference data available
APIA_NO_RESULT	55	No result retrievable
APIA_IMG_NOT_SUPPORT	56	Image format not supported
APIA_VALUE_NOT_DETECTED	60	Value was not found on check image

Error code	Value	Description
APIA_NOT_IMPLEMENTED	70	Invalid call or not implemented
APIA_ALREADY_INIT	71	API already initialized
APIA_NO_API_INIT	72	API not yet initialized
APIA_API_INIT_FAILED	73	API could not be initialized
APIA_NO_SETUP	74	Feature not yet initialized
APIA_ALREADY_SETUP	75	Feature already initialized
APIA_SETUP_FAILED	76	Feature could not be initialized
APIA_FEATURE_INVALID	77	Feature not implemented
APIA_NO_VER_INIT	78	Analysis not yet set up
APIA_VER_INIT_FAILED	79	Analysis could not be set up
APIA_ENGINE_ERR	80	Other error returned from engine
APIA_ERR_PAR	81	Parameter error
APIA_INSUFFICIENT_MEM	82	Not enough memory to fulfil task

Image types

There are several image types possible that need to be taken into account for different possible evaluation features. This applies to item data and reference data as well. The major image types are defined below.

Image type	Value	Description
APIA_IMG_FRONT	0	Front image of a check
APIA_IMG_BACK	1	Back image of a check
APIA_IMG_SNIPPET	2	Signature Snippet Image
APIA_IMG_BANKLOGO	3	Snippet Image of Bank Logo
APIA_IMG_ACCOUNTLOGO	4	Snippet Image of Account Holder Logo
APIA_IMG_BARCODE	5	Snippet Image of Barcode
APIA_IMG_SERIAL	6	Snippet Image of Serial Number
APIA_IMG_CAR	7	Snippet Image of Courtesy Amount Field
APIA_IMG_TEMPLATE_SINGLE	8	Template containing parameters for one image only
APIA_IMG_TEMPLATE_MULTIPLE	9	Template containing parameters for a number of images

Chapter 2

API calls

These are the definitions of the API calls to be implemented by an engine DLL.

Initialization calls

The first set of calls refers to the initialization of engine or engine feature.

Engine initialization principles

The initialization procedure calls should enable the calling application to minimize preparation-overhead if necessary. Whatever is needed for all possible transactions and will not or should not be changed through-out a configuration life-cycle, must not be done for every transaction but rather once during initialization. The first call always might be a general environment initialization done during workspace creation. Parameters passed during this call will overwrite the default parameters of the engine read from property files or set by default. Further calls might change parameters for dedicated features and/or analysis steps but after rest the workspace will be defaulted to the state that was valid after workspace creation.

Workspace creation

Function call

```
__stdcall int APIA_Initialise(int * pnWorkspaceId, tAPIA_InitParms *
pInitialisationParameters)
```

Description

This is always the first call to the API. It initializes the environment and should only be called once. The function checks for the presence of required files and if required, checks license clearance. Parameters passed during this call will overwrite default settings of the engine and will be handled as the new default after successful completion of this call. This could be important for parameter resets after a processing action.

The function creates a unique workspace id into pWorkspaceId if successful. The environment and the workspace will be cleared through APIA_Terminate(…).

Memory handling

Memory for the parameter is allocated by the calling application. The engine is supposed to make a copy of the parameters for further usage. After function call return there is no guarantee that this memory can be further accessed.

Parameter structures

```
typedef struct sAPIA_Parm {
   char *   szParmName;     /* Parameter Name */
   char *   szParmString;   /* Parameter Value*/
} tAPIA_Parm;

typedef struct sAPIA_InitParms {
void (__stdcall * callback)(DWORD nLevel, char *szMsg);   /* callback function for
 tracing
int                     nParmNo;           /* number of parameters */
tAPIA_Parm *            pInitParms;        /* Optional API Init Params, null for empty
 list */
int                     nFeatureParmNo;    /* number of feature parameters */
tAPIA_FeatureParm *     pInitFParms;       /* Optional Feature Init Parameters that must
 be set once, null for empty list */
char *                  szLicenseKey;      /* Opt. License Key String, empty string for no
 license information */
} tAPIA_InitParms;
```

Parameter specification

Name	Type	Description
pnWorkspaceId	Out	Pointer to an Integer value. The function is supposed to return an Integer ID identifying the new workspace. This ID will be used by the calling application for further access to the workspace.
pInitialisationParameters	In	Pointer to tAPIA_InitParms structure that contains all initialization parameters.

Parameter reinitialization

Function call

```
__stdcall int APIA_ResetParameters(int nWorkspaceId)
```

Description

This call reinitializes the environment. The function resets all API parameter to the default values defined in configuration files or by the engine itself. This also includes resetting of all parameter changes done for one or more features. Usually this function is called if the calling application wants to reset parameters for a dedicated workspace to their default state.

Memory handling

Memory for the parameter is allocated by the calling application. The engine is supposed to make a copy of the parameters for further usage. After function call return there is no guarantee that this memory can be further accessed.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace identifier. Parameters of exactly this workspace must be reset.

Workspace release

Function call

```
__stdcall void APIA_Terminate(int nWorkspaceId)
```

Description

This call terminates the API and releases allocated space. It must be the last call to the API.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace identifier of the workspace to be terminated.

Version retrieval and information calls

Version and other information about the engine need to be made available through the version retrieval and information calls to enable the calling environment to provide information on the current version and settings to tracing and monitoring modules.

Version retrieval

Function call

```
__stdcall int APIA_GetVersion(char * szAPI_Version, int * pnVersionLen ,int *
pnFeatureNo, tAPIA_FeatureDesc *pFeatures )
```

Description

This call returns the version description of the API in general and a version description of all features the API implements.

Memory handling

The calling application allocates the memory for the version string and for a default number of tAPIAFeatureDesc records as well. The actual number of allocated records is passed in the pnFeatureNo field. The allocated length of the version string is passed in the pnVersionLen field. If the number of allocated feature structures is insufficient to return all feature descriptions, the engine returns the error code APIA_INSUFFICIENT_MEM. The number of feature descriptions actually available is returned in the pnFeatureNo parameter.

If the allocated version string is insufficient to return the version information, the engine again returns the error code APIA_INSUFFICIENT_MEM. The requested byte length of the version string is returned in the pnVersionLen parameter (includes terminating zeros).

Parameter structures

```
typedef struct sAPIA_FeatureDesc {
int    nFeatureId;            /* Feature ID */
char   szFeatureName[128];    /* Name of Feature */
char   szFeatureType[128];    /* Type of Feature */
char   szFeatureVersion[128]; /* Optional feature version string */
} tAPIA_FeatureDesc;
```

Name	Type	Description
szAPI_Version	Out	Pointer to a character string. The engine is supposed to copy the version info into this string. The string has to be terminated by a null character.
pnVersionLen	In/Out	Pointer to an Integer value that contains the allocated memory for the ApiVersion string. If the engine wants to give back a larger version string as allocated this parameter contains the required length of the version string after function return. The calling application can use this value to allocate the proper amount of memory.
pnFeatureNo	In/Out	Pointer to an Integer value that contains the number of allocated feature description records on input. If the engine wants to give back more feature descriptions as allocated this parameter contains the number of actually available features after function return. The calling application can use this value to allocate the proper amount of memory.
pFeatureDesc	Out	Pointer to an array of feature descriptions. The engine is supposed to copy all relevant feature version information for each feature into the allocated structures. Strings must be terminated by null characters.

Error retrieval

Function Call

```
__stdcall int APIA_GetErrorText(int nError, char * szErrorTextString, int
*pnSize)
```

Description

This call returns the extended error information for the given error code. All available Error Codes must be made public in the vendor's specific APIA header file. The nErrrorTextSize parameter contains the number of characters the calling application has allocated for the error message. The engine has to copy the error text into the provided buffer.

This call returns APIA_OK if successful or a negative error code on failure as defined in header file. If the error code is APIA_NO_MEMORY the engine is not able to return the complete message because the calling application has not allocated enough memory. The text length is returned again in parameter pnSize. In this case the calling application has to retry the call after allocation of sufficient memory.

Memory handling

The calling application allocates a default size for the error string. This number is passed in the pnSize field. If this size is less than the actual size of the error text the engine returns the error code APIA_INSUFFICIENT_MEM. The actual text size is returned in the pnSize parameter. The calling application has to retry the call after allocating sufficient memory.

Parameter specification

Name	Type	Description
nErrorNo	In	The error number returned by a previous function call.
szErrorTextString	Out	Pointer to a null terminated string that gives a textual description of the error.
pnSize	In/Out	Pointer to an Integer value that contains the number of characters the calling application has allocated on input. If this is not sufficient memory to provide the complete error text the parameter contains the actual size of the error message on output.

Tracing calls

Tracing principles

In order to have a consolidated log that assures that all log information will be in correct time order the calling application provides a call-back function that has to be called for each type of tracing information the engine wants to write to the APIA log. The logging level will be set from the calling application using either the initialization call or passing it to the engine with the APIA_SetTraceLevel call. Each time the engine wants to write a trace message to the log the application has to call the APIA_TracePrintf function passing the trace level and the trace information.

Trace level setting

Function call

```
__stdcall void APIA_SetTraceLLevel(int nLevel)
```

Description

This call notifies the engine about a change in the trace level. The engine has to provide tracing information according to the level to the calling application via the APIA_TracePrintf call.

Parameter specification

Name	Type	Description
nLevel	In	Trace level as defined above

Tracing

Function Call

```
void (__stdcall * callback) (int nLevel, char *szMessage)
```

Description

To provide the possibility of having a consolidated trace the calling application provides a CALLBACK function that has to be called for each type of tracing information the engine wants to write to the

APIA trace log. The address of the call-back function will be provided within APIA_Initialise() in the init parameters.

Memory handling

Parameters are only valid within the function call.

Parameter specification

Name	Type	Description
nLevel	In	Trace severity level as defined above
szMessage	In	Message

Engine setup calls

Engine setup principles

Setup calls initialize the API environment for specific GIA features. I.e. initialization needs that are not covered in the API initialization or needs the flexibility to be configured at process time, can be performed through these calls. Setup calls allow the API to use different pre-processing sets for different GIA features. It is also possible to change such setups for one feature during runtime.

Similar to the initialization procedure, the setup calls should enable the calling application to minimize preparation-overhead if necessary. Whatever is needed for all transactions or may be needed for at least multiple transactions in a row should not be done for every transaction but rather fewer times, i.e. only with a setup call.

Preparations that are not really good for multiple transactions and/or with a necessity not easily identified by the calling application, should not be within the setup since this could lead to the need to call the setup for every transaction.

Feature setup

Function call

```
__stdcall int APIA_FeatureSetUp(int nWorkspaceId, int nFeatureNo,
tAPIA_FeatureParm * pFeatureParameters)
```

Description

This call loads setup parameters that are specific for the engine identified by the engine number such as image pre-processing (like de-skew, reverse, clean …). This setup has the character of initialization but can be changed during run time. It is the vendor’s choice to have a default setup to fall back if setup is not performed. The number of allocated parameters is provided in nFeatureNo parameter. It is up to the calling application to call this function for all features at once or to call it for different features subsequently.

Memory handling

Memory for the parameter is allocated by the calling application. The engine is supposed to make a copy of the parameters for further usage. After function call return there is no guarantee that this memory can be further accessed.

Parameter structure

```
typedef struct sAPIA_FeatureParm{
int      nFeatureId;    /* Feature Id */
char *   szParmName;    /* Parameter Name */
char *   szParmString;  /* Parameter Value*/
} tAPIA_FeatureParm;
```

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace identifier in which features will be set up
nFeatureNo	In	Number of features
pFeatureParameters	In	Pointer to an array of feature parameter structures

Feature reset

Function call

```
__stdcall int APIA_ResetFeature(int nWorkspaceId, int nFeatureId)
```

Description

This call resets all parameter done for the feature during previous APIA_FeatureSetup calls. All the values are set to the default values defined in configuration files or by the engine itself.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace identifier in which features will be set up
nFeatureId	In	Identifier for the feature to reset
pFeatureParameters	In	Pointer to an array of feature parameter structures

Analysis setup calls - Item setup

Analysis setup principles

Before any image analysis steps can be performed there are certain preparation steps necessary. The most important of course is the transfer of the image data of the item in question. Another call must provide the transfer of reference images plus optional parameters that need to be considered for a specific reference.

To simplify API calls it is assumed that there is only one item possible per workspace. Therefore there will be no separate item setup calls for different features. Nevertheless, if a validation features actually consists of a certain number of sequential sub-validation there might be a benefit of initiating the item information in the workspace sub-sequentially. This could save costly image transfers.

Item data setup

Function call

```
__stdcall int APIA_LoadItemData(int nWorkspaceId, tAPIA_Data * pItemData)
```

Description

This call is used to hand over additional data of the verification item to the engine's workspace.

Memory handling

The calling application allocates memory for the data structure. This memory will remain valid until the application calls APIA_ClearItem.

Parameter structure

The structure defined here is used for both, item information and reference information as well. In case of detection results it contains all the findings an engine made for a feature. The fields within this structure cover all fields having any value in fraud detection. Features may use only subsets of the fields. Thus not all fields must be filled for each feature, each reference or item. All strings referenced in the structure below must be null terminated.

```
typedef struct sAPIA_Data {
char   szDocRefNo[30+1];   /* Bank number/Market-Code */
char   szBNO[3+1];         /* Bank number/Market-Code */
char   szAccountNo[34+1];  /* Account Number */
char   szBranchNo[20+1];   /* Branch Number */
unsigned long ulAmount;    /* Amount */
char   szCurrency[3+1];    /* Check currency in ISO format*/
char   szSerialNo[30+1];   /* Serial Number */
char   szClearDate[10+1];  /* Clearing date in ISO format 'YYYY-MM-DD' */
char   szRTN[10+1];        /* Route Transit Number */
char   szFormtype[5+1];    /* Item Form type [e.g. IRD=4] */
char   szBankName[100+1];  /* The name of the bank */
char   szAccHolder[300+1]; /* Account Holder Name */
char   szPayeeName[100+1]; /* Payee Name */
char * szFreeText;         /* Unspecified Item Information, only applicable if
 structure is used as input parameter for to the verification engine, e.g a list of
 payee names */
} tAPIA_Data;
```

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
pItemData	In	Pointer to the data structure

Item image setup

Function call

```
__stdcall int APIA_AddItemImages(int nWorkspaceId, int nImageNo, tAPIA_Image *
pImages)
```

Description

This call is used to hand over the image(s) of the verification item to the engine's workspace. The images will be handed as Tiff images.

Additional images also can be added later to the structure to keep image transfer costs low. This is meant to be a call to enable images only to be loaded if a corresponding validation step is possible.

Memory handling

The calling application allocates memory for the image structures. This memory will remain valid until the application calls APIA_ClearItem.

Parameter structure

The structure defined here is used for item information and reference information as well.

```
typedef struct sAPIA_Image {
int    nImageType;     /* Image Type (front, back, snippet, etc. )*/
char   szName[20+1];  /* optional image name/key */
int    resolution     /* Image resolution, can be used as additional information for to
 the verification process, 0 if not relevant */
int    width    /* Image width, can be used as additional information for to the
 verification process, 0 if not relevant */
int    height   /* Image height, can be used as additional information for to the
 verification process, 0 if not relevant */
int    xPos     /* horizontal position, can be used as additional information for to
 the verification process, 0 if not relevant */
int    yPos     /* vertical, can be used as additional information for to the
 verification process, 0 if not relevant */
int    nImageBufferSize;    /* size of image */
const BYTE * pImageBuffer;  /* Image */
} tAPIA_Image;
```

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
nImageNo	In	Number of images
pImages	In	Pointer to the array of images

Item removal

Function call

```
__stdcall int APIA_ClearItem(int nWorkspaceId)
```

Description

This call provides the capability to remove a previously loaded item from the workspace. This function has to be called before switching to a new item. This call is necessary to do a verification of an item with reference data of previous verifications without reloading the reference data.

Memory handling

Memory allocated for this item will be released by the calling application after successful return of this function call.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier

Reference setup calls

Reference setup principles

Comparisons can be done for one single reference or for a set of reference items as well. References will be loaded subsequently by calling the APIA_LoadReference() function. A unique identifier will be provided for each reference by the engine. This identifier can be used to load further images to the reference for additional evaluation steps. With each reference setup call the API provides the ability to pass additional reference data (e.g. the payees name, etc.) to the engine if necessary. Furthermore it is possible to pass reference data that is valid for all references. This can be done with the APIA_LoadRefernceData call.

Global reference data setup

Function call

```
__stdcall int APIA_LoadReferenceData(int nWorkspaceId, tAPIA_Data *
pReferenceData)
```

Description

This call is used to hand over reference data to the engine’s workspace that is valid for all references that will be loaded in further processing steps. Data that is relevant for one reference only will be handed over within the respective APIA_LoadReference call. This call is used to hand over reference data to the engine’s workspace that is valid for all references that will be loaded in further processing steps. Data that is relevant for one reference only will be handed over within the respective APIA_LoadReference call.

Memory handling

The calling application allocates memory for the image structures. This memory will remain valid until the application calls APIA_ClearReferences or APIA_ClearReference.

Parameter structure

See Item data setup.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
pReferenceData	In	Pointer to the data structure

Reference setup

Function call

```
__stdcall int APIA_LoadReference(int nWorkspaceId, int * pnRefId, int
nImageNo, tAPIA_Image * pImages, tAPIA_Data * pReferenceData)
```

Description

This call is used to hand over the image(s) of the verification item to the engine's workspace. The images will be handed as Tiff images.

This call is the first step for a reference item that involves the items image and can be contain reference data as well. The here referred vector of images must only contain real images on the places where it is necessary for the current verification status. Images can be added later to the structure to keep image transfer costs low.

The function returns a reference identifier that can be used to add further images to the reference. This identifier is also used to identify the verification result for a sole reference within a set of results.

Memory handling

The calling application allocates memory for the image structures. This memory will remain valid until the application calls either APIA_ClearReference for the respective reference or APIA_ClearReferences to clear all references.

Parameter structure

See Item data setup and Item image setup.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
pnRefId	Out	Identifier created for the reference by the engine after successful return
nImageNo	In	Number of Ireference images
pImages	In	Pointer to an array of image structures
pReferenceData	In	Pointer to the data structure that belongs to this reference only

Adding reference images

Function call

```
__stdcall int APIA_AddReferenceImage(int nWorkspaceId, int nRefId, tAPIA_Image
* pImage)
```

Description

This call is used to hand over an additional image/snippet of the reference item to the engine's workspace for a specific place in the already existing image vector in the workspace. This is meant to be a call to enable images only to be loaded if a corresponding validation step is possible.

This step requires the inauguration of a reference into the workspace via APIA_LoadReference. The nRefId parameter is the identifier returned by the previous APIA_LoadReference call.

Memory handling

The calling application allocates memory for the image structure. This memory will remain valid until the application calls either APIA_ClearReference for the respective reference or APIA_ClearReferences to clear all references.

Parameter structure

See Item image setup.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
nRefId	In	Identifier of the reference to which the image has the to be added
pImage	In	Pointer to the image structure

Reference removal

Function call

```
__stdcall int APIA_ClearReference(int nWorkspaceId, int nRefId)
```

Description

This call provides the capability to remove previously loaded reference image(s) and data belonging to that reference. This function is for the special purpose to remove one dedicated reference from the workspace. The nRefId parameter is the value returned from APIA_LoadReference when the reference has been added to the workspace. To remove all references the APIA_ClearReferences call has to be used.

Memory handling

After successful completion of this function the calling application does no longer guarantee the validity of images and data structures allocated for this reference in previous verification steps. Usually the calling application will release this memory.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
nRefId	In	Identifier of the reference that has to be removed

Reference clearing

Function call

```
__stdcall int APIA_ClearReferences(int nWorkspaceId)
```

Description

This call provides the capability to remove all previously loaded reference images and data. This includes data that belong to all references as well. This function usually has to be called before switching to a new item. Exception could be if the new item has to be checked with exactly the same references. In this case it will increase performance not to load references again.

Memory handling

After successful completion of this function the calling application does no longer guarantee the validity of images and data structures of all references allocated in previous verification steps. Usually the calling application will release this memory.

Parameter specification

Nam	Type	Description
nWorkspaceId	In	Workspace Identifier

Analysis execution calls

Analysis execution principles

Regarding analysis the API has to distinguish between two main feature types:

- Verification/validation features and
- Detection/recognition features

A verification feature does a comparison of items with references and provides verification results.
A detection feature searches for any data, e.g. the payee name, the address block or even a logo or signature on a given item. The result of this search is provided within the result set. In case of detection features either the field tAPIA_Data or tAPIA_Image must be populated depending on the type of feature. In case of a verification feature these fields will remain empty.

The engine is responsible for memory handling of the respective result sets. The memory will be allocated before result retrieval and has to be released first after the APIA_ClearResults call.

The result of a single engine within a highly integrated fraud detection environment is not a final decision but a component of several findings, which will have to be evaluated in a holistic approach. Therefore the engines are not called to simply return a PAY/No-PAY decision but rather a detailed evaluation result. In case multiple references are made available to an engine, there has to be a result array providing all corresponding results.

Analysis execution

Function call

```
__stdcall int APIA_PerformTest(int nWorkspaceId, int nFeatureNo, int
*pnFeatureId, int *pnResultNo, tAPIA_Result **ppResults)
```

Description

This call actually performs the test(s) for the requested features and with the references added to the workspace.

Function returns APIA_OK if successful or an error code on failure. Failure in this sense is a complete failure. If one of multiple verifications fails, such detailed error must be provided in the detailed error code within the result set array returned by result retrieval functions. Both kinds of error codes are defined in header file.

Memory handling

Memory for result structures will be allocated by the engine. This memory has to remain valid until the application calls the APIA_ClearResults function to explicitly release this memory.

Parameter structure

```
typedef struct sAPIA_Result {
int   nFeatureId; /* Feature Identifier */
int   nRefId;     /* Reference Identifier, if applicable */
int   nRefIndex;  /* Index pointing to matching image in multi-image-templates, if
 applicable */
int   nConfLevel; /* Confidence Level */
int   nErrorCode; /* Error code in case of error */
int   nImageNo;   /* Number of returned images, only for detection features */
tAPIA_Image *  pImages;   /* Images detected by the feature */
tAPIA_Data *   pData;     /* Data detected by the feature */
char*          szComment; /* comment string, e.g. to return additional info */
} tAPIA_Result;
```

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier
nFeatureNo	In	Number of features that have to be performed during this call
pnFeatureId	In	Pointer to an Integer array of the length as specified in nFeatureNo whereby each value represents a feature ID

Name	Type	Description
pnResultNo	Out	Pointer to an integer. The engine has to return the number of returned results (in fact the number of allocated result structures) here.
ppResults	Out	Pointer to a pointer to an array of result structures. The engine has to return all results here.

Result clearance

Function call

```
__stdcall void APIA_ClearResults(int nWorkspaceId)
```

Description

This call forces the engine to release all memory allocated for result passing. The calling application has to make sure that this function is called first after all results have to be examined and probably copied into the own workspace.

Memory handling

After successful completion of this function the engine has no longer to guarantee the validity of result structures allocated in previous verification steps. Usually the engine will release this memory.

Parameter specification

Name	Type	Description
nWorkspaceId	In	Workspace Identifier

Calling sequences

From the experience of Kofax in large projects there seem to be no case where multiple item data sets were involved in a single verification step.

For every verification, validation or detection cycle there are three steps:

1. Verification preparation
   Within this step optional setup or re-setup of verification with process configuration can be done. Further verification is prepped with the corresponding reference information if applicable.
2. Verification execution and result retrieval
   This executes the verification, validation or detection together with handing over of item data. Result data is allocated by the engine and handed over to the calling application.
3. Result set release
   Call for clean up of result structures.

The vendor’s DLL must implement all different call types and utilise the invalid-call return code for all calls that are not applicable in the specific case.

Result scores

For the raw engine results, the scores, the engines should provide a detailed match rate or confidence level in the range of 0 to 100.

Kofax would like the vendors to adjust their engines to produce scores according to following scheme:

Thresholds	Detail result	Rating	Description
AA	100	Maximum Pass	**Maximum level of similarities found based on the evaluated graphic features. / Maximum level of confidence reached for returned validation result or detected value.**
A1	95-99	High Pass - Pass	**High level of similarities found based on the evaluated graphic features. / High level of confidence reached for returned validation result or detected value.** Ratings are usually defined acceptable if supported by other features.
A2	90-94		
A3	86-89		
A4	82-85		
A5	78-81		
B1	74-77	Pass - Caution	**Reasonable level of similarities found based on the evaluated graphic features. / Reasonable level of confidence reached for returned validation result or detected value.** Attention should be administered with ratings as the levels within this category vary depending on the clients operating environment. Ratings are generally acceptable, lower grade can mostly be assigned to image issues gather than fraud.
B2	71-73		
B3	68-70		
B4	65-67		
B5	62-64		
C1	60-61	Caution - Fail	**Unacceptable levels of similarities found based on the parameters extracted. / Unacceptable low level of confidence reached for returned validation result or detected value.** Ratings are never acceptable even if the true reason is image quality unless image quality issues can be automatically detected.
C2	58-59		
C3	56-57		
C4	54-55		
C5	52-53		
D1 - F5	50-51	Fail	Unacceptable levels of similarities found based on the parameters extracted. / Unacceptable low level of confidence reached for returned validation result or detected value. Ratings are never acceptable even if the true reason is image quality unless image quality issues can be automatically detected.
	0-49		

The above table is a sample. The vendor is free to provide a table according to the above scheme with different levels and interpretation ranges as long as these can be somehow mapped into the same threshold logic.