OpenM1 API

Open Source

Reference Guide

1.0

6/10

Table of Contents

1 Overview

The OpenM1 library is designed to greatly simplify the programmatic reading, writing and editing of biometric information packages based on the INCITS standards defined by *M1.3 - Biometric Data Interchange Formats*, such as, just to name a few

- *ANSI INCITS 378-2004, Finger Minutia Format for Data Interchange*
- *ANSI INCITS 381-2004, Finger Image-Based Data Interchange Format*
- *ANSI INCITS 385-2004, Face Recognition Format for Data Interchange*

OpenM1 can equally deal with the European standards' counterparts, defined by the ISO JTC 1/SC 37, such as, for example,

- *ISO/IEC 19794-2, Biometric Data Interchange Formats - Part 2: Finger Minutiae Data*
- *ISO/IEC 19794-4, Biometric Data Interchange Formats - Part 4: Finger Image Based Interchange Format*
- *ISO/IEC 19794-5, Biometric Data Interchange Formats - Part 5: Face Image Data*

Furthermore, OpenM1 can work with any standard that defines file packages composed of multiple sections of data blocks, such as the ones above, by defining an XML configuration file for that standard following the simple OpenM1 rules.

Currently OpenM1 is available for Win32 platforms and is provided as a standard C DLL, an include file for compilation and a LIB file to link against. The library is available in 4 flavors: Debug non-UNICODE, Release non-UNICODE, Debug UNICODE and Release UNICODE. Several C++ code samples demonstrating basic functionality are also available.

2 General Approach

The OpenM1 approach is founded on the realization that most file-format standards are based on the same concept: they are formed of multiple *sections* containing *items* of information and possibly other sections.

In general, we define an ***M1Item*** as a container of data, and an ***M1Section*** as a container of M1Sections and M1Items. We define an ***M1Container*** as being either an M1Section or an M1Item. In programming jargon, M1Section and M1Item both inherit from M1Container. Hence a data package can have the following structure, for example:

```
M1Section
    M1Section
        M1Item
        M1Item
        M1Section
            M1Item
    M1Section
        M1Item
    M1Item
```

The main section is made up of 3 containers, of which the first two are sections and the last is an item. The first of these two sections contains three containers, namely two items and a section, which in turn contains only an item. The second of these two sections also only contains an item.

By using the abstraction of the ***container*** we can easily navigate any tree structure by enumerating all containers one-level below the current container. If we determine the container is a section we recurse into it, and if we determine the container is an item we can deal with the data it contains.
recursing on all containers contained within. M1Items are the leaf nodes of any OpenM1 tree.

This recursion over containers is the foundation of the OpenM1 API.

3 Standard Configuration Files

The format of the standard configuration files are described in XML, based on the schema file ***BDIF.xsd***. It mimics the nested list of sections and items in XML by using <section> and <item> elements. Taking the above scenario as an example, we would have the following general XML structure

```
<section>
    <section>
        <item>
        </item>
        <item>
        </item>
        <section>
            <item>
            </item>
        </section>
    <section>
        <item>
        </item>
    </section>
    <item>
    </item>
</section>
```

On top of this structure, these XML elements also have child elements, which describe the sections and items, such as:

- Sections and items must have a <name>
- Sections and items can have a <description>
- Sections and items can have a <minoccurrences> as well as a <maxoccurrences>, as a means to describe that they can appear multiple times in sequence in a standard package. Lacking these elements it is assumed these containers occur exactly once.
- Sections and items can have a <reserved> element. If set to <true>, the OpenM1 caller creating a package need not provide data for this item or section: OpenM1 sets ***reserved*** containers automatically.
- Items must have a <format> describing the item's datatype, which can be bit, byte, word, dword, tword, string, binary, jpegorjpeg2000 or image.
- Items can have a <min> and <max> value, to be applied to numerical data, or in the case of a string or a binary datatype, these values will describe the length of the data.
- Items can have default values, specified by using the <default> element.
- Items can have a set of values that the value must take on its value from. This is specified by using a series of <listitem>s containing the allowed <name>/<value> pairs.

To define this tree structure in memory we use a nested list of ***M1ContainerDef*** object. An ***M1ContainerDef*** *is an abstract type for an* ***M1SectionDef*** *or an* ***M1ItemDef***. These objects mirror the M1Container, M1Section and M1Item types described above. In essence the types defined above are intended to contain the data, whereas the ****Def*** types are meant to define the data format.

The XML standard configuration file can also contain ***macros*** such as ***length()*** and ***occurrences()*** which enable the standard to define fields which contain the number of bytes or the number of occurrences of particular containers. This facilitates the creation of packages for the user of OpenM1 since these fields are automatically set by OpenM1.

4 Functions

The OpenM1 API provided access to the following functions.

Standard Configuration

- M1ReadStandard
- M1FreeStandard
- M1GetHeadContainer
- M1GetContainerInfo
- M1EnumContainerReset
- M1EnumContainerGetNext
- M1GetItemDefInfo

Data Access

- M1CreatePackage
- M1FreePackage
- M1ReturnSection
- M1SetItemString
- M1SetItemNumber
- M1SetItemBit
- M1SetItemImage
- M1GetItemString
- M1GetItemNumber
- M1GetItemImage
- M1Free

Validation and Serialization

- M1ValidatePackage
- M1WritePackage
- M1ReadPackage

4.1 Standard Configuration

The following functions enable reading and verifying an XML standard configuration file and enumerating the complete structure and all properties within.

For an example of the use of these functions, see the OpenM1 sample ***OpenM1Sample1***.

M1ReadStandard

Read a standard configuration file into memory.

```
int M1ReadStandard(
     const TCHAR*   szPath,
     M1Standard**   ppStandard,
     int            nMaxError,
     TCHAR*         szError
);
```

Parameters

szPath	Input path of standard con to load
ppStandard	Returned pointer to the newly created M1Standard object
nMaxError	Maximum number of characters for error message
szError	Pre-allocated space to receive any error message

Details

The XML standard configuration file is parsed and read into memory. If any XML parsing errors occur a detailed error message is returned.
The standard must be freed with ***M1FreeStandard***.

Return Codes

M1_OK	Success
M1_FAIL	Failed. Check returned error
M1_INVALIDPARAM	One of the pointer parameters was NULL
M1_MEMORYFAULT	Generic memory fault
M1_EXCEPTION	Unexpected error

M1FreeStandard

Frees the M1Standard object allocated by a previous call to ***M1ReadStandard***.

```
int M1FreeStandard(
     M1Standard* pStandard
);
```

Parameters

pStandard	Pointer to the M1Standard object to release

Return Codes

M1_OK	Success
M1_INVALIDPARAM	pStandard is NULL
M1_EXCEPTION	Unexpected error

M1GetHeadContainer

Returns the top-level containerdef for the given standard.

```
int M1GetHeadContainer(
     M1Standard*         pStandard,
     M1ContainerDef**    ppContainer
);
```

Parameters

pStandard	Pointer to standard
ppContainer	Returned containerdef

Details

This is the first step to enumerating all information within a given standard. Note that the containerdef does not need to be release by the caller. It will get released with the standard via the call to ***M1FreeStandard***.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters was NULL
M1_EXCEPTION	Unexpected error

M1GetContainerInfo

Gets information pertaining to a given containerdef.

```
int M1GetContainerInfo(
     M1ContainerDef*     pContainer,
     bool*               pbIsItemDef,
     TCHAR**             pszName,
     TCHAR**             pszDescription,
     TCHAR**             pszMinOccurrences,
     TCHAR**             pszMaxOccurrences,
     bool                *pbIsReserved
);
```

Parameters

pContainer	Input containerdef
pbIsItemDef	Is the containerdef an itemdef or a sectiondef?
pszName	Returned name of container
pszDescription	Returned description of container
pszMinOccurrences	Returned minimum occurrences string for the containerdef
pszMaxOccurrences	Returned maximum occurrences string for the containerdef
pbIsReserved	Is the containerdef reserved?

Details

This function returns all information that is common to both ***sectionsdefs*** and ***itemdefs***. The boolean return value pbIsItemDef is used to determine whether the containerdef is a sectiondef or an itemdef. Note that all strings are pre-allocated by OpenM1 and will get released with the standard when ***M1FreeStandard*** is called.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters was NULL
M1_EXCEPTION	Unexpected error

M1EnumContainerReset

Resets the internal enumeration counter of the containerdef. Used in conjunction with ***M1EnumContainerGetNext*.**

```
int M1EnumContainerReset(
     M1ContainerDef*     pContainer
);
```

Parameters

pContainer	Input containerdef

Details

After it has been ascertained that a containerdef represents a sectiondef, call this function to initiate iteration across all contained containerdefs.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	pContainer is NULL or represents an itemdef
M1_EXCEPTION	Unexpected error

M1EnumContainerGetNext

Iterates through all containerdefs belonging to a given containerdef.

```
int M1EnumContainerGetNext(
     M1ContainerDef*     pContainer,
     M1ContainerDef**    ppContainerNext
);
```

Parameters

pContainer	Input containerdef, must be a sectiondef
ppContainerNext	Next containerdef returned from the iteration

Details

Before calling this function, ***M1EnumContainerReset*** must be called on the containdef to reset its internal iterator. Then typically this function will be called iteratively until it returns M1_FAIL, which signals the end of the list.

Return Codes

M1_OK	Success, *ppContainerNext points to the next containerdef
M1_FAIL	End of list
M1_INVALIDPARAM	pContainer is NULL or represents an itemdef
M1_EXCEPTION	Unexpected error

M1GetItemDefInfo

Retrieves information pertinent to an itemdef.

```
int M1GetItemDefInfo(
     M1ContainerDef*     pContainer,
     M1ItemFormat*       pItemFormat,
     TCHAR**             pszMinValue,
     TCHAR**             pszMaxValue,
     TCHAR**             pszDefaultValue
);
```

Parameters

pContainer	Input containerdef, must be an itemdef
pItemFormat	Returned format of the item
pszMinValue	Returned min string
pszMaxValue	Returned max string
pszDefaultValue	Returned default value string

Details

Once it has been ascertained that a containerdef is an itemdef, this function can be called on it to retrieve the item definition, such as its format and optional minimum, maximum or default values.

Return Codes

M1_OK	Success, *ppContainerNext points to the next containerdef
M1_INVALIDPARAM	pContainer represents a sectiondef or one of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

4.2 Data Access

The OpenM1 data access functions allow the creation of a data package from scratch, to be then validated and written to file. It also allows for the loading from file of an existing data package, with subsequent enumeration of all its data.

Packages can also be modified and re-validate and re-saved, allowing for full editing functionality.

M1CreatePackage

M1CreatePackage allocates space for a new data package and returns a pointer to its root section.

```
int M1CreatePackage(
     const TCHAR*        szPackageName,
     M1Section**         ppHeadSection
);
```

Parameters

szPackageName	Input name of package to create
ppHeadSection	Returned root section of the standard

Details

The specified package name should match that of the standard one wants to create a package for, as in the standard configuration file (*e.g.*, "INCITS 385-2004"). To avoid memory leaks the returned head section must eventually be freed with a call to ***M1FreePackage***.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1FreePackage

Frees a package allocated with ***M1CreatePackage***.

```
int M1FreePackage(
     M1Section*     pHeadSection
);
```

Parameters

pHeadSection	Pointer to package as returned by ***M1CreatePackage***

Return Codes

M1_OK	Success
M1_INVALIDPARAM	The pHeadSection parameter is NULL
M1_MEMORYFAULT	An invalid or previously freed pointer was passed
M1_EXCEPTION	Unexpected error

M1ReturnSection

Returns a pointer to the child section within the given parent section, given the child section's name and index.

```
int M1ReturnSection(
     M1Section*     pParentSection,
     const TCHAR*   szSectionName,
     int            iSection,
     M1Section**    ppSection
);
```

Parameters

pParentSection	Pointer to existing parent section
szSectionName	Input name of child section desired
iSection	Input index of child section desired
ppSection	Returned section

Details

For simplicity of the OpenM1 API, ***M1ReturnSection*** will always return a child section with the given section name and section index: if the section already exists as a child of the parent section then it will be returned and the function will return M1_OK. If on the other hand the section does not exist then it will be created and returned and the function will return M1_SECTIONNOTEXIST. This means that both M1_OK and M1_SECTIONNOTEXIST are to be considered successful return values.
The section should never be freed. It will be de-allocated when the root section is freed with ***M1FreePackage***.
Note that all indices in the OpenM1 API are 1-based, and when specifying multiple occurences of sections the indices should increase sequentially.

Return Codes

M1_OK	Success, indicating the section already existed
M1_SECTIONNOTEXIST	Success, indicating the section was added
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1SetItemString

This function sets a named item within the provided section with a string value.

```
int M1SetItemString(
     M1Section*     pSection,
     const TCHAR*   szItemName,
     int            iItem,
     const TCHAR*   szValue
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
szValue	Input value to set to the specified item

Details

For simplicity of the OpenM1 API, ***M1SetItemString*** will always set the value (assuming the parameters are valid). If the item already exists as a child of the section then it's existing value will be changed to the string provided, otherwise the item will be created and its value subsequently set.
Note that all indices in the OpenM1 API are 1-based, and when specifying multiple occurences of items the indices should increase sequentially.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1SetItemNumber

This function sets a named item within the provided section with a dword value.

```
int M1SetItemNumber(
     M1Section*          pSection,
     const TCHAR*        szItemName,
     int                 iItem,
     int                 iValue
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
iValue	Input value to set to the specified item

Details

For simplicity of the OpenM1 API, ***M1SetItemString*** will always set the value (assuming the parameters are valid). If the item already exists as a child of the section then it's existing value will be changed to the number provided, otherwise the item will be created and its value subsequently set. Note that all indices in the OpenM1 API are 1-based, and when specifying multiple occurences of items the indices should increase sequentially.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1SetItemBit

This function sets a named item within the provided section with a dword value of 0 or 1 based on the provided boolean parameter.

```
int M1SetItemBit(
        M1Section*          pSection,
        const TCHAR*        szItemName,
        int                 iItem,
        bool                bValue
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
bValue	Input value to set to the specified item

Details

This function is only provided for coding simplicity. It just calls ***M1SetItemNumber*** with either 1 or 0 as its ***iValue*** parameter.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1SetItemImage

This function sets a named item within the provided section with a binary value.

```
int M1SetItemImage(
     M1Section*          pSection,
     const TCHAR*        szItemName,
     int                 iItem,
     BYTE*               pImage,
     int                 iImageLen
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
pImage	Input image to set to the item
iImageLen	Input length of pImage

Details

This function behaves like the other ***M1SetItem**** functions in that it either replaces an exiting item or creates it. M1SetItemImage makes a copy of the image, which gets freed when the root section gets freed via ***M1FreePackage***.

Return Codes

M1_OK	Success
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1GetItemString

Returns the string stored in the section's item specified by name and index.

```
int M1GetItemString(
     M1Section*          pSection,
     const TCHAR*        szItemName,
     int                 iItem,
     TCHAR**             pszValue
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
pszValue	Output string

Details

If the item exists, the function sets the pre-allocated string in ****pszValue*** and returns M1_OK, otherwise the function returns M1_ITEMNOTEXIST.
Note that even if the item exists but is not stored as a string, this function will return M1_ITEMNOTEXIST: it is the responsibility of the caller to know the general type of the item's data.

Return Codes

M1_OK	Success
M1_ITEMNOTEXIST	The section does not contain an item (of the appropriate datatype) that goes by the specified name and index
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1GetItemNumber

Returns the number stored in the section's item specified by name and index.

```
int M1GetItemNumber(
     M1Section*          pSection,
     const TCHAR*        szItemName,
     int                 iItem,
     int*                piValue
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
piValue	Output number

Details

If the item exists, the function sets ***piValue*** and returns M1_OK, otherwise the function returns M1_ITEMNOTEXIST.
Note that even if the item exists but is not stored as a number, this function will return M1_ITEMNOTEXIST: it is the responsibility of the caller to know the general type of the item's data.

Return Codes

M1_OK	Success
M1_ITEMNOTEXIST	The section does not contain an item (of the appropriate datatype) that goes by the specified name and index
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1GetItemImage

Returns the image stored in the section's item specified by name and index.

```
int M1GetItemImage(
     M1Section*          pSection,
     const TCHAR*        szItemName,
     int                 iItem,
     BYTE**              ppImage,
     int*                piImageLen
);
```

Parameters

pSection	Pointer to existing section
szItemName	Input name of child item desired
iItem	Input index of child item desired
ppImage	Returned pointer to image
piImageLen	Returned image length in bytes

Details

If the item exists, the function sets ****ppImage*** and ****piImageLen*** and returns M1_OK, otherwise the function returns M1_ITEMNOTEXIST.
Note that even if the item exists but is not stored as an image, this function will return M1_ITEMNOTEXIST: it is the responsibility of the caller to know the general type of the item's data.
The returned image now belongs to the caller, who must call ***M1Free*** to release it.

Return Codes

M1_OK	Success
M1_ITEMNOTEXIST	The section does not contain an item (of the appropriate datatype) that goes by the specified name and index
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1Free

Frees a generic OpenM1 memory block. Currently only to be used on memory allocated by ***M1GetItemImage***.

```
int M1Free(
     BYTE*          pImage
);
```

Parameters

pImage	Input pointer to valid image

Return Codes

M1_OK	Success
M1_INVALIDPARAM	The provided pointer parameters is NULL
M1_EXCEPTION	Unexpected error

4.3 Validation and Serialization

The real heart of OpenM1 is in its validation. During this phase, OpenM1 will try to match up the ***sections*** and ***items*** specified by the caller with the ***sectiondefs*** and ***itemdefs*** specified in the standard configuration.

The 1st pass of the OpenM1 validation process is the called the ***structure adjustment pass***, whereby a copy of the user's data is made piece by piece to allow for careful correlation with the standard's structure:

1. For each containerdef in the standard's definition tree, it is first checked to see if it's reserved. Reserved containers are set by OpenM1 and not by the user, so if the container is reserved, the item or section is explicitly added (recursively) to the section copy, ***sectionAdjusted***. Otherwise all containers in the user data that have the same name are collected and placed in a separate list, ***containersNamed***.

2. If ***containersNamed*** is empty the user has not provided this container, and the code checks to see if there's a default value. If so, the container is added to ***sectionAdjusted***, otherwise an error is flagged and the process ends. This allows the user to skip setting fields that already have appropriate default values in the standard definition.

3. ***containersNamed*** is then checked against the occurrence limitations defined in the associated ***containerDef***, and an error is flagged and the process ends if any conditions have been breached.

4. **containersNamed** is sorted by index, and the provided indices are checked to make sure they are indeed sequential and start at 1. Otherwise an error is flagged and the process ends.

5. If the container is an item, it is added it to ***sectionAdjusted***.

6. If the container is a section, it is recursed back into, using this section and it's associated sectionDef as parameters to continue the process one level down

Once this structure adjustment pass is complete, ***sectionAdjusted*** contains a tree structure that has been validated against the standard's tree structure, whereby missing and reserved containers have been inserted. To complete the validation process the data within the items must be validated according to the rules imposed by its associated itemdef. This is the job of the 2nd pass that is called the ***data coercion pass***, where:

1. For each item in ***sectionAdjusted*** the datatype of the value provided by the caller is ***coerced*** into

the datatype defined by its itemdef. This step is necessary because OpenM1 does not require the user to know the exact final datatype of each final element. To make a case in point, the following 4 function calls will result in the same final setting for the item “Moustache” of datatype bit:

```
M1SetItemBit(pFeatureMask,    "Moustache", 1, true);
M1SetItemNumber(pFeatureMask, "Moustache", 1, 1);
M1SetItemString(pFeatureMask, "Moustache", 1, "1");
M1SetItemString(pFeatureMask, "Moustache", 1, "true");
```

If the data provided cannot be converted into the final datatype an error is flagged and the validation process ends.

2. Finally the data is checked against the optionally provided min and max conditions. For strings and binary types min and max refer to the lengths of the data. Any error is flagged and the process ends at the first error.

The 3rd and final validation step is the ***macro resolution pass***, where the length() and occurrences() macros are evaluated. The description of this process is beyond the scope of this document, please refer to the source code for an in-depth understanding of this step.

The ***M1ValidatePackage*** function returns a highly detailed log of all actions taken, as well as a detailed error string outlining any validation failures which should greatly simplify correcting the invalid package.

M1ValidatePackage

Validates a root section against a standard, returning a detailed log of the process and any potential errors. Package validation is necessary before writing the package.

```
int M1ValidatePackage(
     M1Section*          pHeadSection,
     M1Standard*         pStandard,
     TCHAR**             pszLog,
     TCHAR**             pszError
);
```

Parameters

pHeadSection	Pointer to root section requiring validation
pStandard	Pointer to standard
pszLog	Returned log of detailed steps taken
pszError	Returned error, if validation fails

Details

If validation succeeds, this function returns M1_OK, otherwise it returns M1_FAIL and sets detailed information in the returned error string. In both cases, success or failure, a detailed log of the steps taken is returned, which can assist in diagnosing invalid packages.

Return Codes

M1_OK	Success
M1_FAIL	Validation failed, check returned error string
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

M1WritePackage

Writes the validated disk package to a disk file.

```
int M1WritePackage(
     M1Section*     pHeadSection,
     const TCHAR*   szFile
);
```

Parameters

pHeadSection	Pointer to root section to write to disk
szFile	Path of file to write

Details

Note that the package must be validated against a standard, otherwise this function returns M1_NOTVALIDATED.

Return Codes

M1_OK	Success
M1_NOTVALIDATED	Package has not passed validation yet
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_FILEERROR	Generic file error
M1_EXCEPTION	Unexpected error

M1ReadPackage

Reads a package from file, using the provided standard as a base structure defining the file.

```
int M1ReadPackage(
     const TCHAR*   szFile,
     M1Section**    ppHeadSection,
     M1Standard*    pStandard,
     TCHAR**        pszError
);
```

Parameters

szFile	Path of file to read
ppHeadSection	Pointer to returned root section
pStandard	Pointer to package's standard
pszError	Potential returned error

Details

If the file is successfully read in, the function sets ***ppHeadSection*** and returns M1_OK, otherwise it sets a detailed error message in ***pszError*** and returns M1_FAIL.
The read-in package must eventually be freed with ***M1FreePackage***.
Note that after reading in a package with ***M1ReadPackage***, its overall structure has been validated against the provided standard, but its contents have not been fully validated. To perform full package validation call ***M1ValidatePackage***.

Return Codes

M1_OK	Success
M1_FAIL	Read failed
M1_INVALIDPARAM	One of the pointer parameters is NULL
M1_EXCEPTION	Unexpected error

5 Samples

OpenM1 provides 3 short C++ samples demonstrating the functionality of OpenM1.

These samples make use of the XML standard configuration files that can be found under the **Standards** folder.

5.1 OpenM1Sample1

OpenM1Sample1 outputs detailed information on a set of XML standard configuration files by traversing the containerdefs recursively.

Functions uses:

M1ReadStandard, M1FreeStandard, M1GetHeadContainer, M1GetContainerInfo, M1GetItemDefInfo, M1EnumContainerReset and M1EnumContainerGetNext

Example output for one one the XML standard configuration files is presented here.

```
Standard "..\Standards\ISO 19794-4.xml":
SectionDef "ISO 19794-4":
....SectionDef "General Record Header":
........ItemDef "Format Identifier": fmt(6) min(3) max(3) def(FIR) R
........ItemDef "Version Number": fmt(6) min(3) max(3) def(010) R
........ItemDef "Record Length": fmt(4) def(length(ISO 19794-4)) R
........ItemDef "Capture Device ID": fmt(3)
........ItemDef "Number of Fingers": fmt(2) min(1) def(occurrences(Finger Image Record)) R
........ItemDef "Scale Units": fmt(2)
........ItemDef "Horizontal Scan Resolution": fmt(3)
........ItemDef "Vertical Scan Resolution": fmt(3)
........ItemDef "Horizontal Image Resolution": fmt(3)
........ItemDef "Vertical Image Resolution": fmt(3)
........ItemDef "Pixel Depth": fmt(2) min(1) max(16)
........ItemDef "Image Compression Algorithm": fmt(2)
....SectionDef "Finger Image Record": maxocc(unbounded)
........SectionDef "Finger Image Header":
............ItemDef "Finger Image Record Length": fmt(4) def(length(Finger Image Record)) R
............ItemDef "Finger Position": fmt(2)
............ItemDef "Count of Views": fmt(2) min(1) max(255)
............ItemDef "View Number": fmt(2) min(1) max(255)
............ItemDef "Finger Image Quality": fmt(2) min(0) max(100)
............ItemDef "Impression Type": fmt(2)
............ItemDef "Horizontal Line Length": fmt(3)
............ItemDef "Vertical Line Length": fmt(3)
........ItemDef "Finger Image Data": fmt(0)
```

5.2 OpenM1Sample2

OpenM1Sample2 creates an ISO 19794-5 Face Recognition package by setting a few mandatory text and number fields and providing a sample image.

Functions uses:

M1ReadStandard, M1FreeStandard, M1CreatePackage, M1ReturnSection, M1SetItemNumber, M1SetItemBit, M1SetItemImage, M1ValidatePackage, M1WritePackage and M1FreePackage.

This is the output for this sample program, which is the log string returned by the ***M1ValidatePackage*** function.

Pass 1: Verifying and adjusting general tree structure...
Adding reserved section "Facial Record Header"
Adding reserved item "Format Identifier" with default value "FAC"
Adding reserved item "Version Number" with default value "010"
Adding reserved item "Length of Record" with default value "length(ISO 19794-5)"
Adding reserved item "Number of Facial Images" with default value "occurrences(Facial Record Data)"
Checking occurrences for 1 container(s) "Facial Record Data": 1..unbounded
Checking indices for container "Facial Record Data", index 1 of count 1
Checking occurrences for 1 container(s) "Facial Information Block": 1..1
Checking indices for container "Facial Information Block", index 1 of count 1
Adding reserved item "Face Image Block Length" with default value "length(Facial Record Data)"
Adding reserved item "Number of Feature Points" with default value "occurrences(Feature Points)"
Checking occurrences for 1 container(s) "Gender": 1..1
Checking indices for container "Gender", index 1 of count 1
Checking occurrences for 1 container(s) "Eye Color": 1..1
Checking indices for container "Eye Color", index 1 of count 1
Checking occurrences for 1 container(s) "Hair Color": 1..1
Checking indices for container "Hair Color", index 1 of count 1
Checking occurrences for 1 container(s) "Feature Mask": 1..1
Checking indices for container "Feature Mask", index 1 of count 1
Checking occurrences for 1 container(s) "Features Specified": 1..1
Checking indices for container "Features Specified", index 1 of count 1
Checking occurrences for 1 container(s) "Glasses": 1..1
Checking indices for container "Glasses", index 1 of count 1
Checking occurrences for 1 container(s) "Moustache": 1..1
Checking indices for container "Moustache", index 1 of count 1
Adding missing item "Beard" with default value "0"
Checking occurrences for 1 container(s) "Beard": 1..1
Checking indices for container "Beard", index 1 of count 1
Adding missing item "Teeth Visible" with default value "0"
Checking occurrences for 1 container(s) "Teeth Visible": 1..1

Checking indices for container "Teeth Visible", index 1 of count 1
Adding missing item "Blink" with default value "0"
Checking occurrences for 1 container(s) "Blink": 1..1
Checking indices for container "Blink", index 1 of count 1
Adding missing item "Mouth Open" with default value "0"
Checking occurrences for 1 container(s) "Mouth Open": 1..1
Checking indices for container "Mouth Open", index 1 of count 1
Adding missing item "Left Eye Patch" with default value "0"
Checking occurrences for 1 container(s) "Left Eye Patch": 1..1
Checking indices for container "Left Eye Patch", index 1 of count 1
Adding missing item "Right Eye Patch" with default value "0"
Checking occurrences for 1 container(s) "Right Eye Patch": 1..1
Checking indices for container "Right Eye Patch", index 1 of count 1
Adding missing item "Dark Glasses" with default value "0"
Checking occurrences for 1 container(s) "Dark Glasses": 1..1
Checking indices for container "Dark Glasses", index 1 of count 1
Adding missing item "Feature Distorting Medical Condition" with default value "0"
Checking occurrences for 1 container(s) "Feature Distorting Medical Condition": 1..1
Checking indices for container "Feature Distorting Medical Condition", index 1 of count 1
Adding reserved item "Feature Mask bit 11" with default value "0"
Adding reserved item "Feature Mask bit 12" with default value "0"
Adding reserved item "Feature Mask bit 13" with default value "0"
Adding reserved item "Feature Mask bit 14" with default value "0"
Adding reserved item "Feature Mask bit 15" with default value "0"
Adding reserved item "Feature Mask bit 16" with default value "0"
Adding reserved item "Feature Mask bit 17" with default value "0"
Adding reserved item "Feature Mask bit 18" with default value "0"
Adding reserved item "Feature Mask bit 19" with default value "0"
Adding reserved item "Feature Mask bit 20" with default value "0"
Adding reserved item "Feature Mask bit 21" with default value "0"
Adding reserved item "Feature Mask bit 22" with default value "0"
Adding reserved item "Feature Mask bit 23" with default value "0"
Checking for extraneous containers in section "Feature Mask"
Checking occurrences for 1 container(s) "Expression": 1..1
Checking indices for container "Expression", index 1 of count 1
Adding missing item "Pose Angle - Yaw" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle - Yaw": 1..1
Checking indices for container "Pose Angle - Yaw", index 1 of count 1
Adding missing item "Pose Angle - Pitch" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle - Pitch": 1..1
Checking indices for container "Pose Angle - Pitch", index 1 of count 1
Adding missing item "Pose Angle - Roll" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle - Roll": 1..1
Checking indices for container "Pose Angle - Roll", index 1 of count 1
Adding missing item "Pose Angle Uncertainty - Yaw" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle Uncertainty - Yaw": 1..1
Checking indices for container "Pose Angle Uncertainty - Yaw", index 1 of count 1

Adding missing item "Pose Angle Uncertainty - Pitch" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle Uncertainty - Pitch": 1..1
Checking indices for container "Pose Angle Uncertainty - Pitch", index 1 of count 1
Adding missing item "Pose Angle Uncertainty - Roll" with default value "0"
Checking occurrences for 1 container(s) "Pose Angle Uncertainty - Roll": 1..1
Checking indices for container "Pose Angle Uncertainty - Roll", index 1 of count 1
Checking for extraneous containers in section "Facial Information Block"
Checking occurrences for 2 container(s) "Feature Points": 0..unbounded
Checking indices for container "Feature Points", index 1 of count 2
Checking indices for container "Feature Points", index 2 of count 2
Adding reserved item "Feature Type" with default value "1"
Checking occurrences for 1 container(s) "Feature Point": 1..1
Checking indices for container "Feature Point", index 1 of count 1
Checking occurrences for 1 container(s) "Horizontal Position": 1..1
Checking indices for container "Horizontal Position", index 1 of count 1
Checking occurrences for 1 container(s) "Vertical Position": 1..1
Checking indices for container "Vertical Position", index 1 of count 1
Adding reserved item "Reserved" with default value "0"
Checking for extraneous containers in section "Feature Points"
Adding reserved item "Feature Type" with default value "1"
Checking occurrences for 1 container(s) "Feature Point": 1..1
Checking indices for container "Feature Point", index 1 of count 1
Checking occurrences for 1 container(s) "Horizontal Position": 1..1
Checking indices for container "Horizontal Position", index 1 of count 1
Checking occurrences for 1 container(s) "Vertical Position": 1..1
Checking indices for container "Vertical Position", index 1 of count 1
Adding reserved item "Reserved" with default value "0"
Checking for extraneous containers in section "Feature Points"
Checking occurrences for 1 container(s) "Image Information": 1..1
Checking indices for container "Image Information", index 1 of count 1
Checking occurrences for 1 container(s) "Face Image Type": 1..1
Checking indices for container "Face Image Type", index 1 of count 1
Adding reserved item "Image Data Type" with default value ""
Checking occurrences for 1 container(s) "Width": 1..1
Checking indices for container "Width", index 1 of count 1
Checking occurrences for 1 container(s) "Height": 1..1
Checking indices for container "Height", index 1 of count 1
Checking occurrences for 1 container(s) "Image Color Space": 1..1
Checking indices for container "Image Color Space", index 1 of count 1
Checking occurrences for 1 container(s) "Source Type": 1..1
Checking indices for container "Source Type", index 1 of count 1
Adding missing item "Device Type" with default value "0"
Checking occurrences for 1 container(s) "Device Type": 1..1
Checking indices for container "Device Type", index 1 of count 1
Adding reserved item "Quality" with default value "0"
Checking for extraneous containers in section "Image Information"
Checking occurrences for 1 container(s) "Image Data": 1..1

Checking indices for container "Image Data", index 1 of count 1
Checking for extraneous containers in section "Facial Record Data"
Checking for extraneous containers in section "ISO 19794-5"
Pass 2: Validating individual items...
Validating item "Format Identifier"
Validating item "Version Number"
Validating item "Length of Record"
Validating item "Number of Facial Images"
Validating item "Face Image Block Length"
Validating item "Number of Feature Points"
Validating item "Gender"
Verified legal name/value "Male"/"1" for item Gender
Validating item "Eye Color"
Verified legal name/value "Gray"/"4" for item Eye Color
Validating item "Hair Color"
Verified legal name/value "Brown"/"4" for item Hair Color
Validating item "Features Specified"
Validating item "Glasses"
Validating item "Moustache"
Validating item "Beard"
Validating item "Teeth Visible"
Validating item "Blink"
Validating item "Mouth Open"
Validating item "Left Eye Patch"
Validating item "Right Eye Patch"
Validating item "Dark Glasses"
Validating item "Feature Distorting Medical Condition"
Validating item "Feature Mask bit 11"
Validating item "Feature Mask bit 12"
Validating item "Feature Mask bit 13"
Validating item "Feature Mask bit 14"
Validating item "Feature Mask bit 15"
Validating item "Feature Mask bit 16"
Validating item "Feature Mask bit 17"
Validating item "Feature Mask bit 18"
Validating item "Feature Mask bit 19"
Validating item "Feature Mask bit 20"
Validating item "Feature Mask bit 21"
Validating item "Feature Mask bit 22"
Validating item "Feature Mask bit 23"
Validating item "Expression"
Verified legal name/value "Neutral"/"1" for item Expression
Validating item "Pose Angle - Yaw"
Validating item "Pose Angle - Pitch"
Validating item "Pose Angle - Roll"
Validating item "Pose Angle Uncertainty - Yaw"
Validating item "Pose Angle Uncertainty - Pitch"

Validating item "Pose Angle Uncertainty - Roll"
Validating item "Feature Type"
Validating item "Feature Point"
Validating item "Horizontal Position"
Validating item "Vertical Position"
Validating item "Reserved"
Validating item "Feature Type"
Validating item "Feature Point"
Validating item "Horizontal Position"
Validating item "Vertical Position"
Validating item "Reserved"
Validating item "Face Image Type"
Verified legal name/value "Basic"/"1" for item Face Image Type
Validating item "Image Data Type"
Verified legal name/value "JPEG"/"0" for item Image Data Type
Validating item "Width"
Validating item "Height"
Validating item "Image Color Space"
Verified legal name/value "24 bit RGB"/"1" for item Image Color Space
Validating item "Source Type"
Verified legal name/value "Static photograph from a digital still-image camera"/"2" for item Source Type
Validating item "Device Type"
Validating item "Quality"
Validating item "Image Data"
Pass 3: Resolving macros...
Resolving macro "length(ISO 19794-5)" = "length" of "ISO 19794-5"
Found container "ISO 19794-5" of byte length 60905
Resolving macro "occurrences(Facial Record Data)" = "occurrences" of "Facial Record Data"
Found 1 occurrences of container "Facial Record Data"
Resolving macro "length(Facial Record Data)" = "length" of "Facial Record Data"
Found container "Facial Record Data" of byte length 60891
Resolving macro "occurrences(Feature Points)" = "occurrences" of "Feature Points"
Found 2 occurrences of container "Feature Points"

5.3 OpenM1Sample3

OpenM1Sample3 reads in an ISO 19794-5 Face Recognition package and outputs a text field, a number field, and extracts the face image and writes it to file.

Functions uses:

M1ReadStandard, M1FreeStandard, M1ReadPackage, M1ReturnSection, M1GetItemString, M1GetItemNumber, M1GetItemImage, M1FreePackage, and M1Free.

This sample opens the package binary and using the ***M1Get**** functions outputs

```
Item "Format Identifier" has value "FAC".
Item "Number of Facial Images" has value "1".
```

and writes the extracted image to “sample.jpg”.