Elecard, Ltd.

Elecard Module Configuration Programmer Guide

Notices

Elecard Module Configuration Programmer Guide

Date modified: March 16, 2007

For information, contact Elecard, Ltd.

Tel: +7-3822-492-609; Fax: +7-3822-492-642

More information can be found at: http://www.elecard.com

For Technical Support, please contact the Elecard Technical Support Team: tsup@elecard.net.ru

CONTENTS

1. Introduction

1.1 Preface

Elecard Module Configuration (EMC) is a unified technology for reading and changing settings of Elecard components (DirectShow® filters, etc.). EMC allows the user to:

- Assign a new value to the component parameter;
- Retrieve the current parameter value;
- Retrieve the readable name of the component parameter;
- Verify whether the parameter can be changed or whether it is read-only;
- Retrieve the default parameter value;
- Retrieve the range and set of values that are valid for the component parameter if some restrictions exist;
- Convert parameter value into the human readable string and vice versa, if it is possible;
- Verify whether the present parameter is supported by the component or not;
- Reset all of the parameters to their default values;
- Save all of the setting values to the registry database;
- Restore all of the setting values from the registry database;
- Subscribe/unsubscribe any client component to the module events notification.

1.2 Using this Guide

1.2.1 Purpose

This document provides a detailed description of the EMC technology and specifies interfaces affiliated with EMC which are necessary for configuration of Elecard components delivered with Elecard SDKs.

1.2.2 Topics Covered

The following lists the topics covered in this document:

- **Section 1: Introduction** – provides a general overview of the EMC technology and describes the purpose of the document and its contents.

- **Section 2: Elecard Module Configuration Overview** – provides general information about the EMC technology.

- **Section 3: IParamConfig Interface Specification** – provides a detailed description of the **IParamConfig** interface and its methods.

- **Section 4: IModuleConfig Interface Specification** – provides a detailed description of the **IModuleConfig** interface and its methods.

- **Section 5: Summary of Error Codes** – defines the standard COM errors used in the **IModuleConfig** interface.

- **Section 6: Using IModuleConfig Issues** – some details of the **IModuleConfig** interface use.

1.2.3 Related Documentation

In order to thoroughly understand the DirectShow® technology, it is strongly recommended that the following documentation is read:

- **Microsoft® DirectShow® described in Microsoft® Platform SDK®** – found at: www.msdn.microsoft.com
- **Elecard SDK User Guide** – included in the SDK package.
- **Elecard Components Reference Manual –** included in the SDK package.
- **Elecard SDK Base Classes Reference Manual** – included in the SDK package.

1.2.4 Technical Support

For technical support, please contact the Elecard Technical Support Team:

tsup@elecard.net.ru

2. Elecard Module Configuration Overview

2.1 Introduction

This section provides general information about the EMC technology.

2.1.1 Definitions

Module is the object which the interface **IModuleConfig** is queried from.

Client is the object which queries the interface **IModuleConfig**.

2.2 General Information

2.2.1 Preface

The current version of EMC includes three interfaces: **IModuleConfig**, **IParamConfig**, and **IModuleCallback**. The **IModuleConfig** and **IParamConfig** interfaces are designed to work with the module parameters. The implementation of the **IModuleCallback** interface allows the client to receive and process a notification about the module events.

The interfaces **IModuleConfig** and **IParamConfig** are implemented in the module object.

IModuleConfig inherits the **IPersistStream** interface which in its turn inherits the **IPersist** interface.

The **IModuleConfig** interface provides access for the reading and writing of the module parameters. All of the module parameters have their own unique GUIDs. The parameter GUID must be known in order to read or write its value. All of the parameter values are transferred via the VARIANT structure that represents a general store for different data types. When the parameter value is read, the VARIANT structure is initialized by a type corresponding to the parameter type. When the parameter value should be modified, the filter receives the VARIANT structure filled by the user. If the type, set in the structure, does not correspond with the parameter type, the filter transforms the VARIANT structure type to the parameter type by means of the Win32 API function **VariantChangeType**. In the case when transformation of the type is impossible, the **IModuleConfig** method does not set a new value and an error code is returned.

When the parameter is modified by the **IModuleConfig::SetValue** method call, the new value is placed in the temporary variable and is not used until the **IModuleConfig::CommitChanges** method is called. While the **CommitChanges** method is calling; all of the settings are checked, their compatibility is examined and, if these operations are successful, new settings are applied. Till the **CommitChanges** method is not called, the module uses old parameter values, and the **IModuleConfig::GetValue** method retrieves the actual module values. This allows changing a great number of parameters for one transaction and subsequent verifying all set values at once.

In order to receive the detailed information about a concrete parameter it is necessary to call

IModuleConfig::GetParamConfig and query the pointer to the **IParamConfig** interface. Via the **IParamConfig** interface all the information about the parameter can be retrieved (see below).

The interface **IModuleCallback** must be inherited by the client class. The implementation of the **OnModuleNotify** method allows the client to subscribe for the notification messages about the module parameters modification. For this, it is necessary to query the module for the **IModuleConfig** interface and to call the **RegisterForNotifies** method with a pointer to the class implementing **IModuleCallback** as a parameter. After this any modification of the module parameters will be accompanied by calling the method **OnModuleNotify** implemented in the corresponding client class. The value of the first parameter of the method **OnModuleNotify** will be equal to the number of elements in the array transferred in the second parameter. The second parameter will contain the pointer to the array of GUIDs corresponding to the modified parameters.

2.2.2 Version Interoperability

EMC interfaces are supported by several Elecard DirectShow® filters. For more detailed information see the corresponding filter description in the Elecard Components Reference Manual.

2.2.3 Ownership of Memory

DirectShow filters are In-Process COM servers. That is why all of the arguments declared as pointers are given in the methods as references and it is not required to free the memory allocated for the pointer from the client-side. However when the VARIANT structure is used it is required to free the memory of the structure elements. In this case the **VariantClear** function of Win32 API is to be used.

2.2.4 Standard Interfaces

As per the COM specification, all methods must be implemented on each required interface.

As per the COM specification, any supported optional interfaces must have all functions within the implemented interface, even if the implementation is only a stub implementation returning E_NOTIMPL.

2.2.5 Null Strings and Null Pointers

Null Strings and Null Pointers are NOT the same. A NULL Pointer is an invalid pointer (0) that will cause an exception if used. A NULL String is a valid (non zero) pointer to a one-character array where the character is NULL (i.e. 0). If a NULL string is returned from a method as an [out] parameter (or as an element of a structure), it must be freed; otherwise the memory containing NULL will be lost. Also note that a NULL pointer cannot be passed as an [in,string] argument due to the COM marshalling restrictions. In this case, a pointer to a NULL string should be passed to indicate an omitted parameter.

2.2.6 Errors and Return Codes

In all cases, ‘E’ error codes will indicate FAILED type errors and ‘S’ error codes will indicate, at least, partial success.

Note*: This document does not describe additional standard Microsoft® COM Interfaces such as* ***IUnknown****,* ***IPersistStream****,* ***IPersist****, and DirectShow interfaces such as* ***IPin****,* ***IBaseFilter*** *used by DirectShow® Filters. For more information regarding these interfaces, read the DirectShow® and COM documentation.*

3. IParamConfig Interface Specification

3.1 Defining IParamConfig Interface

The **IParamConfig** interface is intended to receive the detailed information about a concrete parameter and to adjust its characteristics. This interface must be provided by the component. It may be retrieved from the component only by calling the **IModuleConfig::GetParamConfig** method.

The following section describes methods of the **IParamConfig** interface, their parameters, values and return codes.

Table 1. IParamConfig Interface Methods

IParamConfig Methods	Description
IParamConfig::SetValue	Sets a new parameter value.
IParamConfig::GetValue	Retrieves the current parameter value.
IParamConfig::SetVisible	Sets the flag that can be used as a recommendation to visualize the parameter on the GUI.
IParamConfig::GetVisible	Retrieves the value of the flag that fixes the parameter representation on the GUI.
IParamConfig::GetParamID	Retrieves the parameter ID.
IParamConfig::GetName	Retrieves the parameter human readable name.
IParamConfig::GetReadOnly	Clarifies whether the parameter is read-only or not.
IParamConfig::GetFullInfo	Retrieves the parameter value, human readable meaning, name, read-only and visibility status at one call.
IParamConfig::GetDefValue	Retrieves the default parameter value.
IParamConfig::GetValidRange	Retrieves the range of values that is permitted for the parameter if some restrictions are imposed.
IParamConfig::EnumValidValues	Retrieves the set of values that is valid for the parameter if some restrictions are imposed.
IParamConfig::ValueToMeaning	Converts the value to the human readable string if it is possible.
IParamConfig::MeaningToValue	Converts the human readable string to the corresponding value if it is possible.

3.1.1 IParamConfig::SetValue

```
HRESULT SetValue(
        [in] const VARIANT *pValue,
        [in] BOOL bSetAndCommit
);
```

Description

This method sets a new parameter value.

Parameters

- **pValue** – Pointer to the VARIANT variable with a new parameter value.
- **bSetAndCommit** – Flag that can possess the following values:

Value	Description
OATRUE	The method validates and applies changes immediately.
OAFALSE	The modified value should be verified and applied to the supplied module state by calling the **IModuleConfig::CommitChanges** method.

Return Codes

- **S_OK** - Parameter value is modified successfully.
- **S_FALSE** - Parameter value is modified successfully but and during this process the VARIANT value type was successfully changed to the corresponding type of the parameter.
- **E_POINTER** - *pValue* pointer is not valid.
- **E_UNEXPECTED** - Another error.

Note: *When the value type does not match the required parameter type and it is impossible to convert it to the required parameter type, the error code of Win32 API* ***VariantChangeType*** *function is returned.*

3.1.2 IParamConfig::GetValue

```
HRESULT GetValue(
        [out] VARIANT *pValue,
        [in] BOOL bGetCommitted
);
```

Description

This method retrieves the parameter value.

Parameters

- **pValue** – Pointer to the VARIANT variable for the storage of the parameter value.
- **bGetCommitted** – Flag that can possess the following values:

Value	Description
OATRUE	The method retrieves the committed parameter value (applied to the module state).

OAFALSE	The method retrieves the uncommitted parameter value (not applied to the module state).

Return Codes

- **S_OK** - Parameter value is retrieved successfully.
- **E_POINTER** - *pValue* pointer is not valid.
- **E_UNEXPECTED** - Another error.

***Note:** If the retrieved value is BSTR then memory allocated for the **pValue->bstrVal** string must be freed by the Client by calling the Win32 API function **SysFreeString**.*

*If the retrieved value is a pointer to **IUnknown** or its successor then be sure to release the retrieved interface after use.*

3.1.3 IParamConfig::SetVisible

```
HRESULT SetVisible(
       [in] BOOL bVisible
);
```

Description

This method sets the flag that can be used as a recommendation whether the parameter should be displayed on the GUI.

Parameters

- **bVisible –** Flag that is responsible for the parameter visualization. If *bVisible* is OATRUE, it means that the parameter visualization on the GUI is recommended.

Return Codes

- **S_OK** - Flag is set successfully.

3.1.4 IParamConfig::GetVisible

```
HRESULT GetVisible(
       [out] BOOL* bVisible
);
```

Description

This method retrieves the value of the flag that can be used as a recommendation whether the parameter should be displayed on the GUI.

Parameters

- **bVisible –** Pointer to the BOOL variable for the visibility flag.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER -** *pValue* pointer is not valid.

3.1.5 IParamConfig::GetParamID

```
HRESULT GetParamID(
       [out] GUID* pParamID
);
```

Description

This method retrieves the parameter ID.

Parameters

- **pParamID –** Pointer to the GUID variable or the parameter ID storage.

Return Codes

- **S_OK** – Operation is completed successfully.

3.1.6 IParamConfig::GetName

```
HRESULT GetName(
      [out] BSTR* pName
);
```

Description

This method retrieves the parameter human readable name.

Parameters

- **pName –** Pointer to the BSTR variable for the parameter name.

Return Codes

- **S_OK** - Parameter value is modified successfully.
- **E_POINTER** - *pName* pointer is not valid.

Note: *Memory allocated for the pName string must be freed by the Client by calling the Win32 API function* ***SysFreeString****.*

3.1.7 IParamConfig::GetReadOnly

```
HRESULT GetReadOnly(
       [out] BOOL* bReadOnly
);
```

Description

This method clarifies whether the parameter is read-only or not.

Parameters

- **bReadOnly –** Pointer to the BOOL variable for the read-only flag.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - *bReadOnly* pointer is not valid.

3.1.8 IParamConfig::GetFullInfo

```
HRESULT GetFullInfo(
      [out] VARIANT* pValue,
      [out] BSTR* pMeaning,
      [out] BSTR* pName,
      [out] BOOL* bReadOnly,
      [out] BOOL* pVisible
);
```

Description

This method retrieves the parameter committed value, human readable meaning, name, read-only and visibility status at one call.

Parameters

- **pValue –** Pointer to the VARIANT variable for the storage of the parameter value.
- **pMeaning –** Pointer to the BSTR variable for the parameter meaning.
- **pName –** Pointer to the BSTR variable for the parameter name.
- **bReadOnly –** Pointer to the BOOL variable for the read-only flag.
- **pVisible –** Pointer to the BOOL variable for the visibility flag

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** – At least one pointer is not valid.

Note: *Memory allocated for the pName string and for the pMeaning string must be freed by the Client by calling the Win32 API function* ***SysFreeString****.*

If the value retrieved in the VARIANT variable is BSTR then memory allocated for the ***pValue->bstrVal*** *string must be freed by the Client by calling the Win32 API function* ***SysFreeString****.*

3.1.9 IParamConfig::GetDefValue

```
HRESULT GetDefValue(
      [out] VARIANT* pValue
);
```

Description

This method retrieves the default parameter value.

Parameters

- **pValue –** Pointer to the VARIANT variable for the storage of the parameter default value.

Return Codes

- **S_OK**– Operation is completed successfully.
- **E_POINTER** - *pValue* pointer is not valid.

Note: *If the retrieved value is BSTR then memory allocated for the* ***pValue->bstrVal*** *string must be freed by the Client by calling the Win32 API function* ***SysFreeString****.*

If the retrieved value is pointer to ***IUnknown*** *or its successor then be sure to release the retrieved interface after use.*

3.1.10 IParamConfig::GetValidRange

```
HRESULT GetValidRange(
      [out] VARIANT* pMinValue,
      [out] VARIANT* pMaxValue,
      [out] VARIANT* pDelta
);
```

Description

This method retrieves the range of values that is permitted for the parameter if some restrictions are imposed.

Parameters

- **pMinValue –** Pointer to the VARIANT variable for the parameter minimum value.
- **pMaxValue –** Pointer to the VARIANT variable for the parameter maximum value.
- **pDelta –** Pointer to the VARIANT variable for the parameter increment value (or the decrement value).

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - At least one pointer is not valid.

3.1.11 IParamConfig::EnumValidValues

```
HRESULT EnumValidValues(
      [in][out] long* pNumValidValues,
      [in][out] VARIANT* pValidValues,
      [in][out] BSTR* pValueNames
);
```

Description

This method retrieves the set of values that is valid for the parameter if some restrictions are imposed.

- **pNumValidValues –** Pointer to the long variable. If both of *pValidValues* and *pValueNames* are equal to NULL, *pNumValidValues* returns with the number of valid parameter values. Otherwise, this variable is equal to the number of values to return in arrays pointed by the *pValidValues* and *pValueNames*. When the method returns successfully, this value is the actual number of valid values returned in the *pValidValues* and *pValueNames* arrays.
- **pValidValues –** Pointer to the array of the parameter valid values. If it is not NULL, up to **pNumValidValues* valid values are copied into this array. The size of the buffer pointed by *pValidValues* should be at least **pNumValidValues* multiplied by **sizeof**(VARIANT).

- **pValueNames** – Pointer to the array of names of the parameter valid values. If it is not NULL, up to **pNumValidValues* names of the parameter valid values are copied into this array. The size of the buffer pointed by *pValueNames* should be at least **pNumValidValues* multiplied by **sizeof**(BSTR).

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - At least one pointer is not valid.

***Note:** Memory allocated for each of pValueNames strings must be freed by the Client by calling the Win32 API function **SysFreeString**.*

You may call this method with pValidValues and pValueNames set to NULL to get the required size of arrays.

If one of pValidValues or pValueNames is NULL then the method fills the array which pointer is not equal to NULL.

3.1.12 IParamConfig::ValueToMeaning

```
HRESULT ValueToMeaning(
      [in] const VARIANT* pValue,
      [out] BSTR* pMeaning
);
```

Description

This method converts the parameter value to the human readable string if it is possible.

Parameters

- **pValue –** Pointer to the VARIANT structure in which the parameter value is stored.
- **pMeaning** – Pointer to the BSTR variable for the parameter meaning.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - At least one pointer is not valid.

***Note:** Memory allocated for the pMeaning string must be freed by the Client by calling the Win32 API function **SysFreeString**.*

3.1.13 IParamConfig:: MeaningToValue

```
HRESULT MeaningToValue(
      [in] const BSTR pMeaning,
      [out] VARIANT* pValue
);
```

Description

This method converts the human readable string to the corresponding value if it is possible.

Parameters

- **pMeaning –**BSTR variable with the parameter human readable meaning.

- **pValue –** Pointer to the VARIANT variable for the parameter value.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - At least one pointer is not valid.

Note: *If the retrieved value is BSTR then memory allocated for the* ***pValue->bstrVal*** *string must be freed by the Client by calling the Win32 API function* ***SysFreeString****.*

4. IModuleConfig Interface Specification

4.1 Defining IModuleConfig Interface

The **IModuleConfig** interface is a module configuration interface that enables tuning the module parameters. This interface must be provided by the module and all functions must be implemented.

The **IModuleConfig** interface inherits the **IPersistStream** interface which in its turn inherits the **IPersist** interface.

The ID of the **IModuleConfig** interface is **IID_IModuleConfig**

{0x486F726E, 0x4D43, 0x49b9, {0x8A, 0x0C, 0xC2, 0x2A, 0x2B, 0x05, 0x24, 0xE8}}

The entry in the registry is the following:

HKEY_CLASSES_ROOT\Interface\{486F726E-4D43-49b9-8A0C-C22A2B0524E8} = IModuleConfig

The following section describes methods of the **IModuleConfig** interface, their parameters, values and return codes.

Table 2. IModuleConfig Interface Methods

IModuleConfig Methods	Description
IModuleConfig::SetValue	Sets a new parameter value.
IModuleConfig::GetValue	Retrieves the current parameter value.
IModuleConfig::GetParamConfig	Retrieves the pointer to the **IParamConfig** interface.
IModuleConfig::IsSupported	Clarifies whether the parameter identified by *pParamID* is available for the given module or not.
IModuleConfig::SetDefState	Resets all of the module parameters to default values.
IModuleConfig::EnumParams	Retrieves the list of parameters that are valid for the given module.
IModuleConfig::CommitChanges	Verifies and applies the modified parameter values.
IModuleConfig::DeclineChanges	Declines all of the parameter modifications that have been made since the last **CommitChanges** call; sets the module to the previous committed state.
IModuleConfig::SaveToRegistry	Saves the committed module state into the registry database.
IModuleConfig::LoadFromRegistry	Loads the module parameters from the registry database. The loaded values should be verified and applied by the **CommitChanges** call.
IModuleConfig::RegisterForNotifies	Subscribes the client for the notification messages about the module parameters modification.

IModuleConfig::UnregisterFromNotifies	Unsubscribes the client from the notification messages about the module parameters modification.

4.1.1 IModuleConfig::SetValue

```
HRESULT SetValue(
      [in] const GUID* pParamID,
      [in] const VARIANT* pValue
);
```

Description

This method assigns a new value to the module parameter identified by the *pParamID* unique identifier. This value should be verified and applied to the module internal state by the **CommitChanges** call.

Parameters

- **pParamID –** Pointer to the parameter identifier.
- **pValue –** Pointer to the VARIANT variable with a new value of the parameter identified by *pParamID*.

Return Codes

- **S_OK** - Parameter value is modified successfully.
- **S_FALSE** - Parameter value is modified successfully but during this process the VARIANT variable type was successfully changed to the corresponding type of the parameter.
- **E_POINTER** - *pValue* pointer is not valid.
- **E_UNEXPECTED** - Another error.

Note: *When the value type does not match the required parameter type and it is impossible to convert it to the required parameter type, the error code of the Win32 API* ***VariantChangeType*** *function is returned.*

4.1.2 IModuleConfig::GetValue

```
HRESULT GetValue(
      [in] const GUID* pParamID,
      [out] VARIANT* pValue
);
```

Description

This method retrieves the value of the module parameter identified by the *pParamID* unique identifier.

Parameters

- **pParamID –** Pointer to the parameter identifier.
- **pValue –** Pointer to the VARIANT value for the parameter value.

Return Codes

- **S_OK** - Parameter value is retrieved successfully.
- **E_POINTER** - *pValue* pointer is not valid.
- **E_UNEXPECTED** - Another error.

***Note:** If the retrieved value is BSTR then memory allocated for the **pValue->bstrVal** string must be freed by the Client by calling the Win32 API function **SysFreeString**.*

*If the retrieved value is pointer to **IUnknown** or its successor then be sure to release the retrieved interface after use.*

4.1.3 IModuleConfig::GetParamConfig

```
HRESULT GetParamConfig(
      [in] const GUID* pParamID,
      [out] IParamConfig** pValue
);
```

Description

This method retrieves the pointer to the **IParamConfig** interface of the module parameter identified by the *pParamID* unique identifier.

Parameters

- **pParamID –** Pointer to the parameter identifier.
- **pValue –** Pointer to the variable that receives a pointer to the **IParamConfig** interface of the parameter object identified by *pParamID*. The retrieved value is NULL if the specified parameter is not supported by the module.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_POINTER** - *pValue* pointer is not valid.
- **E_INVALIDARG** - Another error.

***Note:** If the operation is completed successfully then this method increments the reference counter of the queried interface. Therefore be sure to release the retrieved interface after use.*

4.1.4 IModuleConfig::IsSupported

```
HRESULT IsSupported(
      [in] const GUID* pParamID
);
```

Description

This method clarifies whether the parameter identified by *pParamID* is valuable for the present module or not.

Parameters

- **pParamID –** Pointer to the parameter identifier.

Return Codes

- **S_OK** – Parameter identified by *pParamID* is supported by the module.

- **S_FALSE** – Parameter identified by *pParamID* is not supported by the module.
- **E_POINTER** - *pParamID* pointer is not valid.

4.1.5 IModuleConfig::SetDefState

```
HRESULT SetDefState()
```

Description

This method resets all of the module parameters to default values.

Return Codes

- **S_OK** – Operation is completed successfully.
- **E_OUTOFMEMORY –** Operation can not be completed due to insufficient memory.

Note: *This method notifies and applies the default values after reset.*

4.1.6 IModuleConfig::EnumParams

```
HRESULT EnumParams(
      [in][out] long* pNumParams,
      [in][out] GUID* pParamIDs
);
```

Description

This method retrieves the list of parameters that are valid for the present module.

Parameters

- **pNumParams –** Pointer to the long variable. If *pParamIDs* is NULL, *pNumParams* returns with the number of parameters supported by the module. Otherwise, this variable is equal to the number of parameter identifiers to return in array pointed by *pParamIDs*. When the method returns successfully, this value is the actual number of parameter IDs returned in the *pParamIDs* array.
- **pParamIDs –** Pointer to the array of parameter identifiers. If it is not NULL, up to **pNumParams* parameter identifiers are copied into this array. The size of the buffer pointed by *pParamIDs* should be at least **pNumParams* multiplied by **sizeof**(GUID).

Return Codes

- **S_OK** – Operation is completed successfully.

Note: *You may call this method with pParamIDs set to NULL to get the required size of the pParamIDs array.*

4.1.7 IModuleConfig::CommitChanges

```
HRESULT CommitChanges(
      [out] VARIANT* pReason
);
```

Description

This method verifies and applies the modified parameter values to the module internal state.

Parameters

- **pReason –** If the method fails then *pReason* points to the VARIANT variable with the error explanation.

Return Codes

- **S_OK** – Operation is completed successfully.

Note: *Usually* ***pReason->vt*** *is equal to* VT_BSTR *and* ***pReason->bstrVal*** *contains the BSTR error description. But if the method fails, it is better to check whether* ***pReason->vt*** *is equal to VT_BSTR or not, because an error explanation depends on the module realization.*

4.1.8 IModuleConfig::DeclineChanges

```
HRESULT DeclineChanges();
```

Description

This method declines all of the parameter modifications that have been made since the last **CommitChanges** call (i.e. unverified and unapplied changes) and sets the module to the previous committed state.

Return Codes

- **S_OK** – Operation is completed successfully.

4.1.9 IModuleConfig::SaveToRegistry

```
HRESULT SaveToRegistry(
      [in] DWORD hKeyRoot,
      [in] const BSTR pszKeyName,
      [in] const BOOL bPreferReadable
);
```

Description

This method saves into the registry database the module internal state that was successfully applied by the last **CommitChanges** call.

Parameters

- **hKeyRoot** – Handle to an registry root key. This handle is returned by the **RegCreateKeyEx** or **RegOpenKeyEx** functions, or it can be one of the following predefined keys:

  HKEY_CLASSES_ROOT
  HKEY_CURRENT_CONFIG
  HKEY_CURRENT_USER
  HKEY_LOCAL_MACHINE
  HKEY_USERS

- **pszKeyName –** Pointer to a null-terminated string specifying the name of a subkey to save parameters. This subkey must be a subkey of the key identified by the *hKeyRoot* parameter. This parameter cannot be NULL.

- **bPreferReadable –** If TRUE, then the module parameters should be written into the registry as human readable values, otherwise as a binary stream.

Return Codes

- **S_OK** – Operation is completed successfully.

4.1.10 IModuleConfig::LoadFromRegistry

```
HRESULT LoadFromRegistry(
      [in] DWORD hKeyRoot,
      [in] const BSTR pszKeyName,
      [in] const BOOL bPreferReadable
);
```

Description

This method loads the module parameters from the registry database.

Parameters

- **hKeyRoot** – Handle to an registry root key. This handle is returned by the **RegCreateKeyEx** or **RegOpenKeyEx** functions, or it can be one of the following predefined keys:

  HKEY_CLASSES_ROOT
  HKEY_CURRENT_CONFIG
  HKEY_CURRENT_USER
  HKEY_LOCAL_MACHINE
  HKEY_USERS

- **pszKeyName –** Pointer to a null-terminated string specifying the name of a subkey to the read parameters. This subkey must be a subkey of the key identified by the *hKeyRoot* parameter. This parameter cannot be NULL.
- **bPreferReadable –** If TRUE, then the module parameters should be read from the registry as human readable values, otherwise as a binary stream.

Return Codes

- **S_OK** – Operation is completed successfully.

***Note:** The loaded values should be verified and applied by the **CommitChanges** call.*

4.1.11 IModuleConfig::RegisterForNotifies

```
HRESULT RegisterForNotifies(
      [in] IModuleCallback* pModuleEventsCallback
);
```

Description

This method subscribes the client for the notification messages about the module parameters modification.

Parameters

- **pModuleEventsCallback –** Pointer to the client class implementing the **IModuleCallback** interface.

Return Codes

- **S_OK** – Operation is completed successfully.

4.1.12 IModuleConfig::UnregisterFromNotifies

```
HRESULT UnregisterFromNotifies(
      [in] IModuleCallback* pModuleEventsCallback
);
```

Description

This method unsubscribes the client from the notification messages about the module parameters modification.

Parameters

- **pModuleEventsCallback –** Pointer to the client class implementing the **IModuleCallback** interface.

Return Codes

- **S_OK** – Operation is completed successfully.

5. Summary of Error Codes

The following table defines the standard COM errors used in the **IModuleConfig** interface.

Table 3. Definitions of the errors used in the IModuleConfig interface

Standard COM errors used by IModuleConfig	Description
E_FAIL	An unspecified error.
E_INVALIDARG	The value of one or more parameters is not valid. This is generally used in the case of a more specific error when a problem is expected to be unlikely or not easily identified (for example when there is only one parameter).
E_NOINTERFACE	The interface is not supported.
E_NOTIMPL	Not implemented.
E_OUTOFMEMORY	The memory is insufficient to complete the requested operation. This can happen at any time as the server needs to allocate memory to complete the requested operation.

6. Using IModuleConfig Issues

For the correct use of the **IModuleConfig** interface it is required:

- *ModuleConfig.h* – a file describing the **IModuleConfig** interface.
- A file with the GUID descriptions for the supported settings. This file is created by the developer for each filter supporting the **IModuleConfig** interface. Such a file is called **.h*, where [*] is a short filter name (for example, *FilterName.h*).

These header files must be included using the **include** statement.