
INTERFACE SPECIFICATION

isystem.open

Purpose

This technical paper describes the *isystem.open* interface. It is designed to facilitate implementation of client and server side objects.

The *isystem.open* Interface

The *isystem.open* interface was designed to allow easy adaptation and integration of various cross-platform debugging tools into the winIDEA environment that connects to these server objects as a client.

In the design the main goal was to keep the universal, hardware independent code on the client side and let the server object implement only specific functions.

The client should thus implement following functionality:

- The application window and all window types necessary (editor, memory windows, etc.).
- A mechanism to store and restore configuration information
- High level algorithms independent of the server object, like file and project management, third party compiler tool integration, loading symbolic files, high level debugging, profiler, RTOS support, disassemblers, SFR and FLASH database, etc.

The server object would typically implement following:

- Object specific configurations (dialogs)
- If necessary a communication channel to an external hardware device
- Capability to store and restore configuration in a compound file provided by the client
- Low level functions like reading/writing memory, setting breakpoints etc.

Connection Mechanism

Binding

The *isystem.open* interfaces are COM interfaces, however as the server object is a singleton, its interface is obtained by calling an exported function from its module (DLL), rather than by calling *CoCreateInstance*.

Currently the following binding mechanism is used:

- The client application (EXE) binds to a server (DLL) by loading it dynamically using the LoadLibrary API function.
- It then obtains a pointer to a function named ***H_CreateObject*** using the *GetProcAddress* API function. It is servers responsibility to provide (export) such a function.
- Through a call to the ***H_CreateObject*** an *IUnknown* pointer to the server object is obtained.
- Further interfaces are obtained in standard COM fashion via *QueryInterface* calls through the *IUnknown* pointer.
- As of this point, the client will only make calls to the server through the interfaces returned by the server.

Note that the server always runs in-process, because it interfaces the clients user interface elements, like the menu bar. If a part of the server must be implemented as a shared resource, it is suggested that user interface elements are stored in the DLL loaded by the client and non-user interface elements in a separate module (DLL or EXE).

Interfaces

Global Definitions and Types

Globdefs.h

This file defines global definitions also used in other interfaces (other than *isystem.open*).

```
#define MAXBITS 32

// maximum bit support
#define ADDRESS DWORD

// address currently a 32 bit unsigned
#define DATA DWORD

// data (registers) currently a 32 bit unsigned
#define MAU DWORD

// maximum bit size of single addressable unit
#define TRACETIME DWORD

// maximum timer width for the trace interface
```

CAddress

A structure containing both the memory area and offset within.

```
struct CAddress
{
    ADDRESS m_aAddress; // offset
    short   m_iArea;    // memory area as specified in CDebugInterface::GetRegisterAreaInfo
    BYTE    m_byProcess; // index of the virtual process - SBZ
    BYTE    m_byReserved; // reserved - SBZ
};
```

CCPUInfo

Describes the current CPU core.

MemArea.h

This file defines the memory areas for different CPU families.

iOPENDefs.h

This file defines global definitions for *isystem.open* interfaces

Interface Constants

- MAX_IOPEN_NAME_LENGTH

Maximum length (including the terminating zero) of a 'name' – used for register names etc.

- MAX_IOPEN_STRING_LENGTH

Maximum length (including the terminating zero) of a 'string' – used for error messages etc.

- IDM_IOPEN_BEGIN
- IDM_IOPEN_END

The allowed range for server object menu based commands.

Register, memory area and CPU mode

```

struct SiOPEN_Register
{
    TCHAR m_szName[MAX_IOPEN_NAME_LENGTH];
    BYTE  m_bySize;           // size of register in bits
    DWORD m_dwFlags;         // info on the register - CRegisterInfo::EFlags
    DWORD m_dwCPUMode;       // in which CPU modes is this register available
    TCHAR m_cFlags[MAXBITS]; // if it is a flag register, this flags should be displayed.
    // invalid flags must be marked with underscore
};

struct SiOPEN_MemoryArea
{
    BYTE  m_byBitsPerMAU; // number of bits in a MAU
    ADDRESS m_aStart;     // first valid address in the area
    ADDRESS m_aEnd;       // last valid address in the area
    enum EFlags
    {
        flRTRead      = 0x00000001, // area can be accessed in real time
        flRTWrite     = 0x00000002, // area can be accessed in real time
        flCPUExec     = 0x00000004, // CPU can execute from this area
        flCPURead     = 0x00000008, // CPU can read from this area
        flCPUWrite    = 0x00000010, // CPU can write to this area
        flDownload    = 0x00000020, // can download in this area
        flCanProfile  = 0x00000040, // profiler available in this area
        flCanCoverCode = 0x00000080, // code coverage available
        flCanCoverData = 0x00000100, // data coverage available
    };
    DWORD m_dwFlags;
};

struct SiOPEN_CPUMode
{
    TCHAR m_szName[MAX_IOPEN_NAME_LENGTH]; // name of the mode
    DWORD m_dwCPUMode; // with which flag(s) is it identified in CStatus::m_dwCPUMode
};

```

IOpenSystem Interface

The IOpenSystem interface is the only interface that the server must implement. All others are optional.

Its purpose is to provide basic maintenance of the server object and its connection to the client.

HRESULT Attach(const CClientInfo & ClientInfo);

This is the first function called by the client after the binding mechanism has completed.

ClientInfo

This structure conveys client info to the server.

```
struct CClientInfo
{
    HWND m_hwndMainWnd; // handle of the client's main window
    TCHAR m_szDescription[MAX_IOPEN_NAME_LENGTH]; // description of the client
    IOpenClient * m_pIOpenClient; // client interface
    TCHAR m_szParameters[MAX_IOPEN_NAME_LENGTH]; // command line parameters passed
};
```

HRESULT Setup(WORD wFlags);

This function is called upon workspace changes.

wFlags

The wFlags indicates the type of workspace change:

- **sfInit** – used when initializing a new workspace. The server should reset itself to its default settings and if necessary allocate resources needed for the workspace (cache files, etc.).
- **sfDone** – used before closing a workspace. The server should release all resources that it allocated in the sfInit call
- **sfOnNew** – used after a new workspace has been loaded. This gives the object a chance to validate the loaded configuration.

This is a typical call sequence of the Setup function after the client attaches:

```
pISystem->Setup(sfInit); // initialize workspace
```

... the last workspace is loaded ...

```
pISystem->Setup(sfOnNew); // workspace loaded
```

If the user selects a different workspace:

```
pISystem->Setup(sfDone); // close current workspace
```

```
pISystem->Setup(sfInit); // initialize workspace
```

... the new workspace is loaded ...

```
pISystem->Setup(sfOnNew); // workspace loaded
```

When the client exits, but before the Detach function is called, the client will call:

```
pISystem->Setup(sfDone); // close current workspace
```

HRESULT GetInfo(CInfo * pInfo);

The client will call this function to obtain information about the server object.

```
struct CInfo
{
    GUID m_CLSID;
    TCHAR m_szDescription[MAX_IOPEN_NAME_LENGTH]; // description of the debugger
};
```

m_szDescription

a textual description of the server

m_CLSID

A GUID identifying the server object. Client can use this info to be able to handle different servers on the same workspace.

HRESULT CreateMenu(HMENU hMenu, DWORD * pdwInsertPosition);

This function is called whenever the client needs to rebuild the application menu bar.

hMenu

The handle to the already constructed client menu. If this value is NULL, the server should present a modal GUI that provides access to all configuration options otherwise inserted in the application menu bar. In this case the client does not need to call the *DispatchCommand* function.

pdwInsertPosition

This points to an array of DWORDs indicating positions of various submenus in the hMenu.

Currently following are defined:

- *mpSystem* position to insert HW menu
- *mpHelp* help menu position

Note: the inserted command items must be in the range IDM_IOPEN_BEGIN - IDM_IOPEN_END

HRESULT DispatchCommand(DWORD dwCommand, DWORD & dwFlags);

The *DispatchCommand* is called whenever one of the menu items inserted by the server is selected as well as to update the command availability and status.

dwCommand

this is the ID of the command (as inserted in the menu by the *CreateMenu* function)

dwFlags

the input value of this field whether the command was selected or just its status is requested (*duiUpdate* set)

if the *duiUpdate* flag is set, the status of the command, must be checked and returned in the *dwFlags*:

- *duoEnabled* the command is enabled
- *duoChecked* the command has a check mark
- *duoNA* the command is not supported

HRESULT Initialize();

Called to initialize the server's hardware. This is typically called upon a request to start debugging.

HRESULT Poll();

This function is called periodically. It should check if the hardware (initialized by *Initialize*) is still responding.

HRESULT GetErrorMessage (LPTSTR pszError, BYTE byErrorType, LONG & rlErrorID);

After an operation fails, this function is called to obtain error and/or warning message(s).

pszError

the string to fill. Should there be no error or warning pending, this field should be set to an empty string.

byErrorType

Either *etCurrentError* or *etCurrentWarning*, to obtain errors and warnings respectively.

rlErrorID

The numerical value of the error (filled by the server).

HRESULT Serialize(IStorage * pIStorage, BOOL bSave);

Called by the client to store or restore configuration.

pIStorage

pointer to IStorage, where the configuration data is (to be) stored.

bSave

TRUE if configuration data should be saved. Otherwise the data is loaded from the storage.

HRESULT ModifySymbols();

The client will call this function to indicate that symbolic names have changed. The server should iterate all symbolic configurations through *ICallBack::ModifySymbol*

HRESULT OSHelper(BYTE byFunc, PVOID pParams, BOOL & rbResult);

This function is called to keep modeless UI elements of the server.

byFunc

- *ohMessage* *pParams* specifies a windows message to be handled. *rbResult* should return nonzero if the message is handled
- *ohIdle* *pParams* points to a LONG indicating the number of iterations of the idle function after the last message, *rbResult* should return nonzero if the server currently performs background processing in the idle call

MFC Example:

```
BOOL CIOPENClientDemoApp::PreTranslateMessage(MSG* pMsg)
{
    if (CWinApp::PreTranslateMessage(pMsg))
        return TRUE;
    BOOL bResult=FALSE;
    if (NULL!=m_IOPEN.m_pISystem)
        m_IOPEN.m_pISystem->OSHelper(IOpenSystem::ohMessage, pMsg, bResult);
    return bResult;
}

BOOL CIOPENClientDemoApp::OnIdle(LONG lCount)
{
    // first the default
    BOOL bResult=CWinApp::OnIdle(lCount);
    BOOL bsResult=FALSE;
    if (NULL != m_pISystem)
    {
        // next poll the server if already initialized
        m_bServerInitialized = SUCCEEDED(m_pISystem->Poll());
        // let the server handle UI idle actions
        m_pISystem->OSHelper(IOpenSystem::ohIdle, &lCount, bsResult);
    }
    return bResult || bsResult;
}
```

IOpenClient Interface

The IOpenClient interface is used as means of server to client communication. The server can make use of its functions to notify the client of a configuration change, to convert symbols to values etc.

Note: this interface is implemented on the client side. A pointer to it is passed to the server in the ISystem::Attach call. The server can call any of its functions immediately after the Attach call. Implementation of functions of this interface is optional

HRESULT Change(WORD wInterface, DWORD dwFlags);

The server should call this function to indicate a change in one of the interfaces (like when CPU type is changed).

wInterface

Specifies which interface has changed.

```
enum EInterface
{
    idSystem,          // the ISystem interface
    idDebug,           // the IDebug interface
    idTrace,           // the ITrace interface
    idProfiler,        // the IProfile interface
    idDataCoverage,    // the ICoverData interface
    idCodeCoverage,    // the ICoverCode interface
    idPROM,            // the IPROM interface
    idFLASH,           // the IFLASH interface
};
```

dwFlags

Additional information on the nature of change.

```
enum EChangeFlags
{
    cfInterfaceChange = 0, // the interface has changed
    cfDebugMemoryWritten = 1, // memory was written, the client should refresh
};
```

HRESULT Status(WORD wInterface, LPCTSTR pszStatus, DWORD dwFlags);

The client can use this function to notify the client of change in status in one of its interfaces. This way the client can immediately show the new status instead of waiting for the current (usually lengthy) server call to return and update the status using the ReadStatus function.

wInterface

the interface that is notifying the status change.

pszStatus

the status string to be displayed by the client.

dwFlags

one of the following

- *sfAction* regular action performed
- *sfError* error condition

HRESULT ConvertPath(WORD wConversion, LPTSTR pszPath);

This function converts the file path.

wConversion

the type of the conversion to perform.

- *cvToRelative* convert to relative path
- *cvToAbsolute* convert to absolute
- *cvToTarget* convert to (project's) target path

pszPath

the file name. The result is returned in the same field.

HRESULT BrowseSymbol(LPTSTR pszSymbol, DWORD dwMaxLen, DWORD dwFlags);

Through this call the server (typically one of its dialogs) can obtain a symbolic name. The client should give the user an input option (symbol browser dialog) from which a symbol can be selected.

pszSymbol

the field in which the symbolic name is returned. *dwMaxLen* specifies the maximum length of this buffer.

dwFlags

Flags specifying which symbol are applicable

HRESULT ModifySymbol(LPTSTR pszSymbol, DWORD dwMaxLen, PVOID pInfo);

The server should call this function for every of its symbol based configurations when the *IOpenDebugger* interface receives the *ModifySymbols* call.

pszSymbol

the symbol to be modified. The new symbol value is returned in the same field. If the symbol was deleted from the symbol table an empty string is returned.

pInfo

the value passed to the *ModifySymbols* call.

HRESULT Evaluate(LPTSTR pszExpression, DWORD dwMaxLen, CAddress & radrResult, ADDRESS & aSize);

The server can use this function to convert a symbolic name or an expression to an address.

pszExpression

the expression to be evaluated. If there are errors in the conversion, the error string is returned in this field.

radrResult

The address calculated from the expression.

aSize

The size of expression in MAUs. A function size is the difference between its exit and entry points plus one, size of a variable is determined by its type, etc.

HRESULT EvaluateExpression(LPTSTR pszExpression, DWORD dwMaxLen);

The server can use this function to calculate an expression to a string value

pszExpression

the expression to be evaluated. The result or error message is returned in this field.

HRESULT APIOutput(LPCTSTR pszString);

This function outputs a message to the current output destination of the script interpreter.

pszString

The string to be written.

HRESULT SymbolFromAddress (const CAddress & adrAdr, DWORD dwOptions, LPTSTR pszSymbol, DWORD dwMaxLen)

Converts an address to a symbol.

adrAdr

The address to be converted.

dwOptions

Specifies the type of symbols to qualify for conversion.

pszSymbol

The returned symbol.

HRESULT FindFunction (const CAddress & adrAdr, CFunc & rFunc)

Finds a function on the *adrAdr*.

rFunc

Returns the found function information.

HRESULT TrackSource (const CAddress & adrAdr, DWORD dwFlags)

Tracks source or disassembly on the *adrAdr*.

dwFlags

- *tsfSource* track source
- *tsfDisassembly* track disassembly
- *tsfSetFocus* set focus to the tracked window

HRESULT GetMemoryInfo (CMemInfo &rMemInfo)

Client will call this function to investigate properties of a memory location (i.e. endian ordering, thumb, etc).

rMemInfo

- *m_byProp* property of the area as defined in MemArea.h for the current CPU
- *m_adrAddress* address to examine
- *m_aLast* last address of the same property

HRESULT SetStatusBar (int iPane, LPCTSTR pszText, DWORD dwExtra)

Mainpulates the clients status bar.

iPane

- *cwsText* set text only
- *cwsProgressSwitch* *dwExtra* is >0 for show progress
- *cwsProgressValue* *dwExtra* is 0-100

pszText

The new text to display. NULL resets to default.

IOpenDebugger Interface

HRESULT GetInfo(CInfo & rInfo);

The client will call this function after workspace has been loaded or after the server notified the client through the *IOpenClient::Change* function.

rInfo

The information about the (debug) server is passed to the client in this field.

```
struct CInfo
{
    enum EFlags
    {
        fl0StepModeNone      = 0x00000000, // should never occur
        fl0StepModeLines     = 0x00000002, // HW can quickly set BPs on any number of lines
        fl0StepHWBP         = 0x00000008, // hardware breakpoint is available
        fl0ConservativeBP    = 0x00000009, // fl0StepHWBP, breakpoints should be set conservtively
        fl0StepModeMask     = 0x0000000F, // step mode mask

        fl0RTAccess         = 0x00000010, // real time access is available
        fl0EnableBPsBeforeEmu= 0x00000020, // BPs are enabled before emulation is started
        fl0HotAttach        = 0x00000040, // system is configured for hot attaching
        fl0ProvideAccessInfo = 0x00000080, // access info is provided for memory/register access

        fl0RegSize4         = 0x00000000, // register value info is 4 bytes/register
        fl0RegSize8         = 0x00000100, // register value info is 8 bytes/register
        fl0RegSize16        = 0x00000200, // register value info is 16 bytes/register
        fl0RegSize32        = 0x00000300, // register value info is 32 bytes/register
        fl0RegSizeMask     = 0x00000300, // register value size mask

        fl0HWTTaskRunControl = 0x00000400, // can do HW task run control
        fl0HWTTaskTrace     = 0x00000800, // can do HW task trace

        fl0EndianLittle     = 0x00000000, // little endian CPU
        fl0EndianBig        = 0x00001000, // big endian CPU
        fl0EndianMask       = 0x00003000, // endian mask

        fl0CantReset        = 0x10000000, // reset command is not available
        fl0CantSingleStep   = 0x20000000, // CPU can't single step, client must update regs to
allow disassembly

        fl0Initialized      = 0x80000000, // initialized - valid
    };
    DWORD m_dwFlags;
    CCPUIInfo m_CPUInfo; // CPU family by ECPUFamily
    TCHAR m_szCPUName[MAX_HINTERFACE_LONG_NAME_LENGTH]; // unique CPU name - used for SFRs
    WORD m_wNumRegisters; // number of CPUs registers
    BYTE m_byNumAreas; // number of CPUs memory areas
    BYTE m_byNumCPUModes; // number of CPU modes
};
```

HRESULT GetRegisterAreaInfo(SiOPEN_Register * pRegisters, SiOPEN_MemoryArea * pMemoryAreas, SiOPEN_CPUMode * pCPUModes);

The client will call this function after a call to *GetInfo*. The client will allocate enough space (as reported in the *GetInfo* call in *CInfo::m_wNumRegisters*, *CInfo::m_byNumAreas* and *CInfo::m_byNumCPUModes*) for the *pRegisters*, *pMemoryAreas* and *pCPUMode* arrays.

pRegisters

This array describes all CPU core registers. This should not include SFRs.

pMemoryAreas

This array describes all memory areas accessible by the CPU.

pCPUModes

This array describes all CPU modes relevant for the CPU

HRESULT GetAddressString(const CAddress & adrAddress, LPTSTR pszAddress, DWORD dwFlags);

Formats the Address to hexadecimal, memory area appropriate result

adrAddress

The address to format.

pszAddress

The string where the result is returned.

dwFlags

- 0 formats the address
- afsAddressFormat returns the formatting string

HRESULT ReadStatus(CStatus & rStatus);

Reads the current status of the debugger.

rStatus

```
struct CStatus
{
    enum EStatus
    {
        stMustInit, // the debug system must initialize
        stStopped,  // CPU is stopped
        stRunning,  // CPU is running
        stReset,    // CPU is held in reset
        stHalted,   // CPU is halted by target
        stWaiting,  // CPU is halted by emulator
        stAttach,   // debugger is initialized and waiting for hot attachment on the target
    };
    BYTE    m_byStatus;
    DWORD   m_dwCPUMode;
    BYTE    m_byMemProperty; // property of memory at execution point
    CAddress m_adrPC;        // execution point, if m_nStatus==stStopped
    ADDRESS m_aRegBase;     // SFR register base
};
```

HRESULT ReadRegisters(BYTE * pbyValues);

Reads all registers into pbyValues

pbyValues

for every CPU core register, its value must be put in the array. Every register occupies number of bytes specified in *CInfo::fIORegSizeXX*

HRESULT WriteRegister(WORD wRegister, BYTE * pbyValue);

Modifies a register.

wRegister

The index of the register as enumerated in the GetRegisterAreaInfo.

pbyValue

The new value. Size as specified in *CInfo::fIORegSizeXX*

HRESULT ReadMemory(const CAddress & adrAddress, BOOL bRealTime, ADDRESS aNumMAUs, BYTE * pbyBuf);

Reads memory.

adrAddress

address to read from.

bRealTime

TRUE if the operation should be performed without stopping the CPU.

aNumMAUs

The number of MAUs to read. MAUs 1-8 bits will occupy one byte/MAU. 9-16bits 2 bytes, 17-32bits 4 bytes etc.

pbyBuf

the buffer where the memory is read in.

HRESULT WriteMemory(const CAddress & adrAddress, BOOL bRealTime, ADDRESS aNumMAUs, const BYTE * pbyBuf);

Writes memory.

adrAddress

address to write to.

bRealTime

TRUE if the operation should be performed without stopping the CPU.

aNumMAUs

The number of MAUs to write.

pbyBuf

the buffer to write.

HRESULT SetExecutionBreakpoints(BOOL bSet, BYTE byMode, BYTE byArea, DWORD dwNum, const ADDRESS * paAddresses);

Sets or clears execution breakpoints.

bSet

TRUE if breakpoints should be set, FALSE if cleared.

byMode

Specifies what type of breakpoints should be set. According to the step mode specification of the GetInfo, the client will (not)use some types.

- *bpmRegular* regular breakpoints
- *bpmLine* line breakpoints (used only if debugger supports separate execution breakpoint class)
- *bpmStep* single BP used for stepping
- *bpmLineInfo* all line symbols - info only

byArea

The memory area where the breakpoints are to be set.

dwNum

Number of breakpoints to be set.

paAddresses

An array of addresses where the breakpoints are to be set to.

HRESULT CPUReset(BOOL bRun);

Resets the CPU.

bRun

if TRUE, the CPU should be set in running after released from reset., otherwise the CPU should stop before executing the first instruction of the program.

HRESULT CPURun(BOOL bRun, BOOL & bWasRunning, BOOL bLong);

Runs or stops the CPU.

bRun

TRUE if the CPU should be set to running, FALSE if it should stop.

bWasRunning

In this field, the debugger should return the previous status of the CPU. It is possible that the CPU was stopped due to a breakpoint between the last *ReadStatus* call and current stop request.

bLong

The client indicates whether the run is expected to last a long time (run and run-until) operations. If this parameter is FALSE, the run command is a part of a high-level step. In such case, the debugger should disable interrupts before setting CPU to running.

HRESULT CPUGoto(const CAddress & adrAddress);

Presets execution point.

adrAddress

The address where the current execution point is to be set.

HRESULT CPUStep(BOOL bHigh, BOOL bOn);

Performs a step.

bHigh

if FALSE a single instruction step is executed. If TRUE, line breakpoints should be activated and CPU set to running (if also *bOn*)

bOn

If *bHigh*, this field determines whether the high level step mode should be entered or exited.

HRESULT SwitchDownload(BYTE byOn);

Called to indicate emulation mode change.

byOn

Defines the new emulation mode

- *dlNewSession* first call to indicate new session
- *dlBegin* begin of download (CPU put in RESET)
- *dlEnd* end of regular download (CPU released from RESET)
- *dlRepeatBegin* begin of repeated download
- *dlRepeatEnd* end of repeated download
- *dlAttach* hot attach
- *dlDetach* hot detach
- *dlNewSession*

HRESULT SetMemoryProp(BYTE byProp, BYTE byArea, ADDRESS aStart, ADDRESS aEnd)

The client informs the debugger that a specific memory region has special properties.

byProp

The property of the memory region. Uses values defined in *MemArea.h*

byArea

The memory area index.

aStart

The start address of the region.

aEnd

the last address of the region.