SwyxIt! Client SDK 6.02

Swyx Solutions AG
Joseph-von-Fraunhofer-Str. 13a
44227 Dortmund

1 Overview

One strength of SwyxIt! is the possibility of integration into 3rd party software. This allows starting calls from CRM software or popping up screens depending on the caller id. It is even possible to build systems for predictive dialing. SwyxIt! provides two ways of integration into 3rd party applications: TAPI and Client SDK. Both interfaces allow at least basic call control. If there are no special reasons for using TAPI (e.g. an existing TAPI application), we recommend to use the Client SDK. Using Client SDK you may use any call control options of SwyxIt!.
The SDK contains samples in C++, C# and VB. When you have read this documentation, please pay some attention to the sample code, showing some common tasks.

1.1 Disclaimer

You may use the Client SDK on your own risk. The SDK technically allows using the product in a way that is not supported or intended. By doing this some functions might not work as expected.

This Client SDK and the sample code are provided as is. The SDK and the documented API is subject to change without notice. Further development of the product might require changes of the API that can lead to incompatibility with previous versions. We don't guarantee that all currently supported functions are part of future versions. Nevertheless the API is used internally as well and we will not change it without very good reasons.

The Line Manager COM interface has been tested internally using…

- Microsoft Visual Basic 6
- C++ using Visual Studio 6, Visual Studio .NET 2003, Visual Studio .NET 2005
- Visual Basic .NET using Visual Studio .NET 2003, Visual Studio .NET 2005
- C# using Visual Studio .NET 2003, Visual Studio .NET 2005

We don't guarantee that the API works with any future versions of Microsoft Visual Studio.

The Line Manager COM API should be usable from any programming language or tool that can interoperate with COM. But we cannot guarantee that the API works with other programming languages or tools from other vendors like Borland Delphi.

1.2 Operation Mode

1.2.1 SwyxIt! vs. SwyxIt! Now

SwyxIt! can run as a full featured SwyxWare client or as a multi registration SIP client (SwyxIt! Now). Today this is no real configuration option but a difference in the setup packages. Both clients support the Client SDK. But due to the differences some features might be not applicable.

- SwyxIt! connects to a SwyxServer and supports all SwyxWare features like sever based conference, server based music on hold, callback on busy, CTI for SwyxPhone, intercom calls, call intrusion, call routing manager, graphical script editor… The configuration (roaming user profile) is stored on the server and shared between all clients of the same user.
- SwyxIt! Now connects to one or more SIP providers simultaneously. All basic call handling is supported, including call swap, call transfer, call hold, conference calls… The configuration is stored locally.

1.2.2 Normal Mode vs. Power Dial Mode

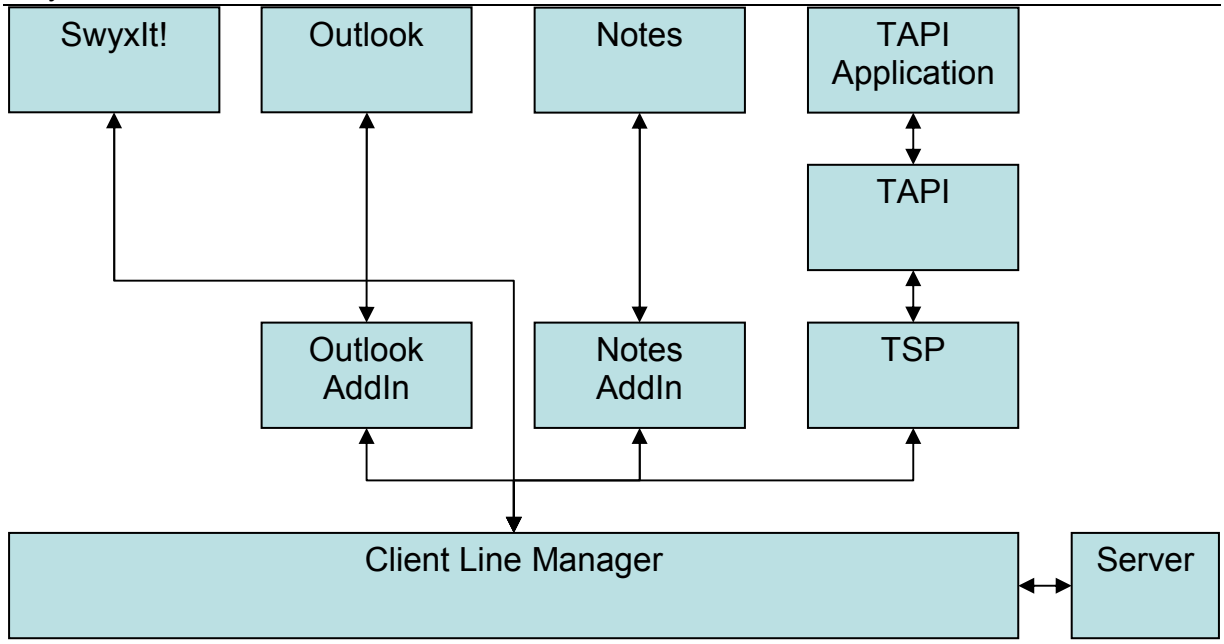
- Typically SwyxIt! runs in normal operation mode. In normal operation mode the audio (voice, dial tone, alerting, busy…) of the selected line (the line in focus) is linked to a local sound device (handset, headset, speakers…). There is always exactly one line selected. When a line is hooked off (e.g. because the user has clicked on it), it becomes the selected line and the previously selected line becomes unselected. Only the selected line can be active (connected to peer, established voice connection, linked to local sound device), the other lines are inactive, on hold, terminated or maybe ringing for an incoming call. When the currently selected line is active and a different line is going to be selected, the previously selected line is automatically put on hold. When the currently selected line is dialing or alerting and a different line is going to be selected, the previously selected line is automatically disconnected.
- The power dial mode is used for implementing automatic call distribution systems, predictive dialing, wake up call scenarios and such. Since this mode is intended only for unattended call handling, there is no linkage to the local sound device at all. As a result it is possible to have multiple active lines. The voice connection is handled by the line, by default it sends silence to the peer and incoming voice from the peer is ignored. There is no automatism related to line selection. When a line is hooked off, it does not become selected. When a line becomes unselected, it is not put on hold or disconnected implicitly. As a matter of fact the line selection has no real meaning in this mode. The only effect of line selection is that the SwyxIt! display shows information for the selected line and user input from SwyxIt! is forwarded to the selected line. But since we are talking about an unattended system, this does not matter. In most cases the SwyxIt! user interface is not running anyway in this mode.

The power dial mode is enabled in the registry:

- Key: “HKLM\SOFTWARE\Swyx\Client Line Manager\CurrentVersion\Options”
  - DWORD value “EnablePowerDialMode”: set to 1 for power dial mode, set to 0 for normal operation mode (default when value does not exist).
  - DWORD value “CancelBlindTransferOnVoicemail”: set to 0 when a blind call transfer should not be cancelled when the transfer call is connected to the voicemail or call routing. By default the client will abort a blind call transfer when it gets connected to call routing since usually we want to transfer someone to a real person. For power dial systems it is very common to transfer calls into scripts, so this automatism should be switched off.

1.3 Client Architecture

The core component of SwyxIt! is the Client Line Manager (CLMgr). All applications like SwyxIt!, Swyx Client TSP, Swyx Outlook AddIn and third party applications are running on top of the Client Line Manager. All these applications are using the same Client Line Manager instance. In this way several applications may use the same lines simultaneously. So when a call is established using a TAPI application, the same call will be visible within SwyxIt!. A call can be initiated by the Outlook AddIn and then be put on hold by TAPI.
The CLMgr connects to the server respective SIP proxy, handles calls and manages audio streams. All other components are accessing the server through the CLMgr.



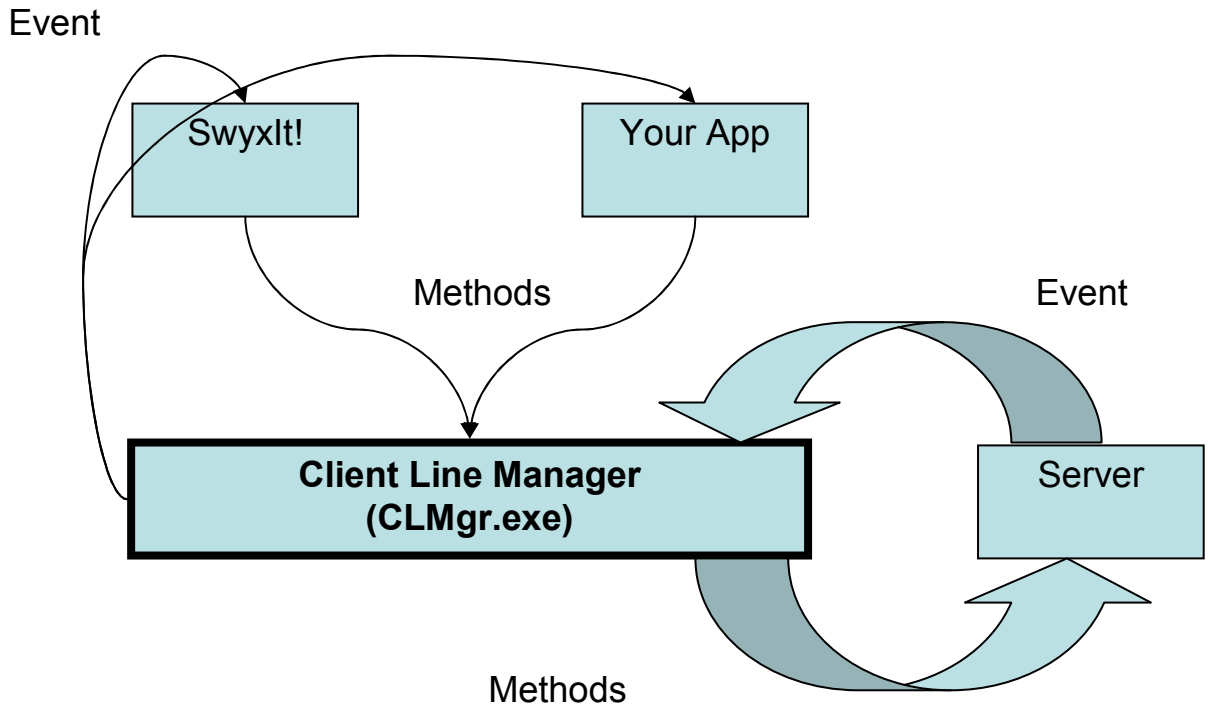
1.4 Methods and Events

The CLMgr provides a set of methods for call control, media streaming and configuration. The API allows tasks like dialing, call hold, call transfer, establishing of a conference… The functions published in the Client SDK allow nearly everything that can be done via SwyxIt! as well. The other way round the CLMgr will send events to the client applications when anything important happens: incoming calls, a call gets connected or disconnected, a speed dial state changes...

Most call related functions are handled asynchronous. This is merely like in old ship movies: The captain tells the mate: “full speed ahead”. The mate tells the cox: “full speed ahead”. The cox tells the engineer: “full speed ahead”. The engineer finally pushes the button and confirms it to the cox and the message queue goes back till the captain.

Example: When an incoming call is accepted in the SwyxIt! user interface, SwyxIt! calls the appropriate CLMgr function. The CLMgr handles that task asynchronously and tells the server to accept the call. When that request is confirmed by the server, the call gets accepted; the CLMgr connects the line, starts the media streaming and tells SwyxIt! that the line state has been changed from “ringing” to “active”. Finally SwyxIt! displays the line in connected state and updates the display.

The important message is: The implementation has to be aware that call related API calls are handled asynchronously. When e.g. an invalid number is dialed, the dial function will return without error but later on the line will get disconnected with an appropriate disconnect reason.

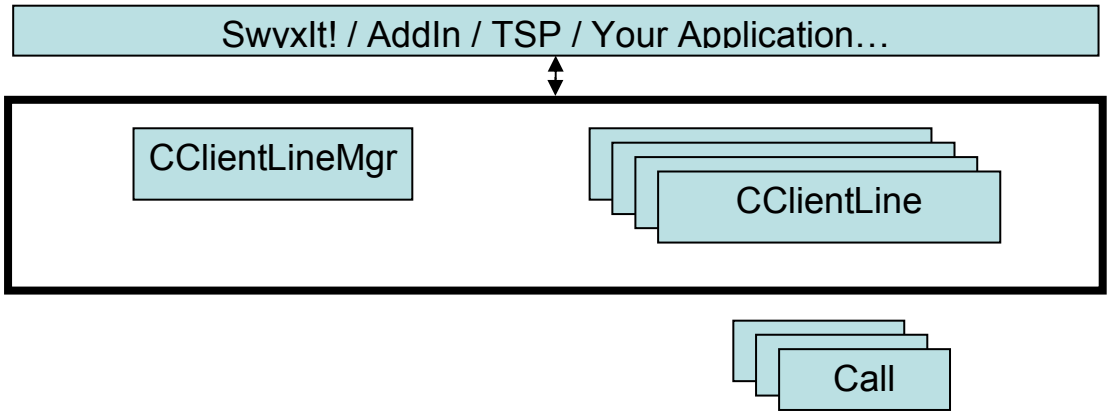


1.5 Lines, Selected Line & Line States

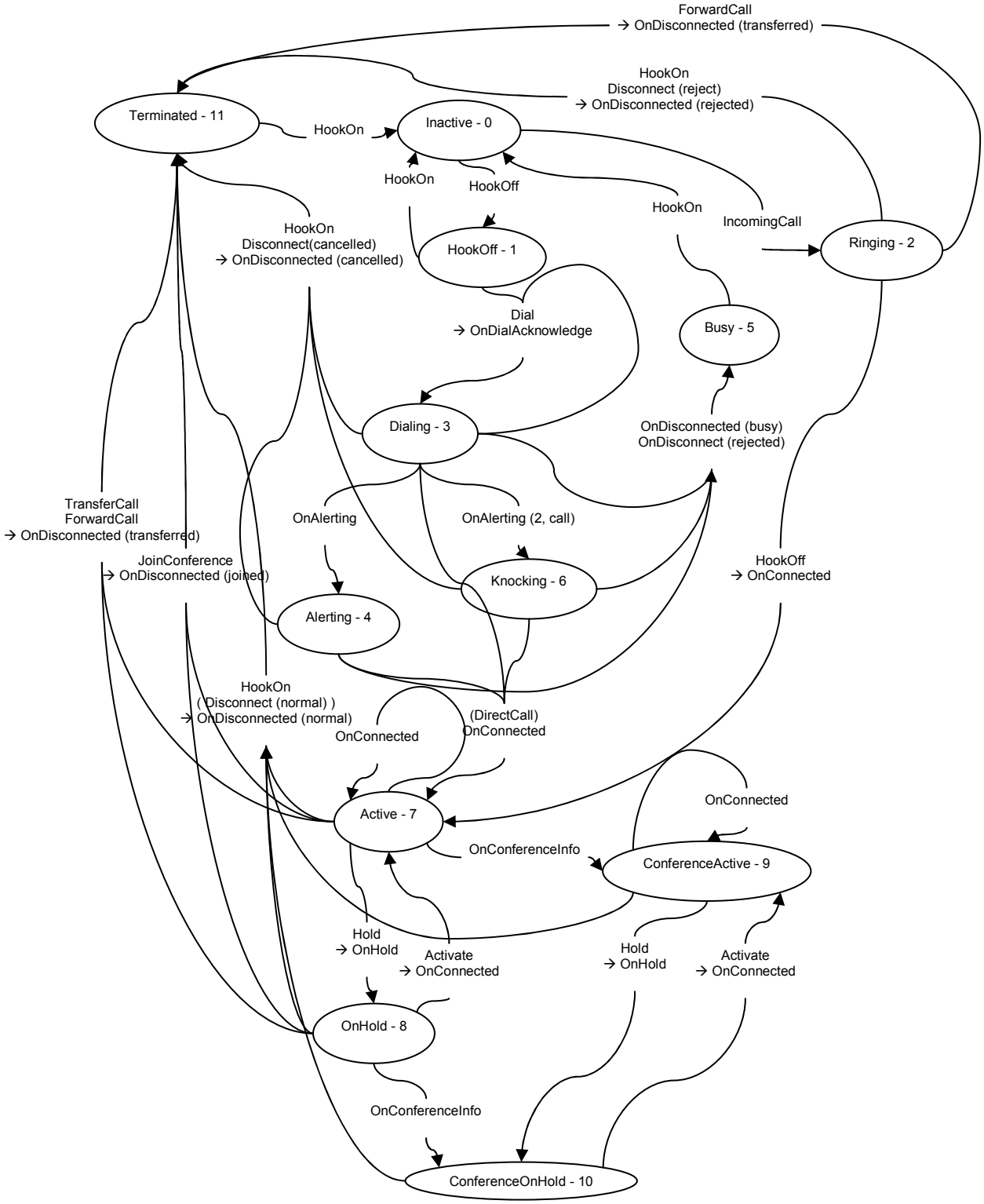
Within the line manager we have one CClientLineMgr object and multiple CClientLine objects. The job of the CClientLineMgr object is reading and writing configuration, registration with the server or SIP registrars, audio device handling, switching between lines… The CClientLine objects are handling the individual calls. There is always one line selected (line in focus). The selected line is linked to the audio device. In normal operation mode it is not possible having two or more active calls. The CClientLineMgr will ensure that an active line is automatically put on hold when a different line becomes active. A line is always in one of the following states, resembling the state of the corresponding call:

- **Inactive**: line is idle, no call
- **HookOffInternal**: outgoing call, hook or line clicked, handset off hook, internal dial tone is played. Waiting for more digits.
- **HookOffExternal**: outgoing call, line is off hook and the public access prefix has been dialed so far. Waiting for more digits.
- **Ringing**: the line is signaling an incoming call
- **Dialing**: outgoing call, one or more digits have been dialed so far. Waiting for more digits or a response from the server.
- **Alerting**: outgoing call, the called party (destination, peer) will hear ringing
- **Knocking**: outgoing call. This is a secondary call for the peer, the peer is already talking on an other line.
- **Busy**: outgoing call, the destination is busy
- **Active**: incoming or outgoing call, logical and physical connection are established
- **OnHold**: incoming or outgoing call, the peer gets music on hold
- **ConferenceActive**: incoming or outgoing call, a conference with multiple participants has been established.

- **ConferenceOnHold**: incoming or outgoing call, a conference with multiple participants has been established but we have currently put the conference on hold. The conference participants will not hear music on hold.
- **Terminated**: incoming or outgoing call, the call has been disconnected actively or passively.
- **Transferring**: incoming or outgoing call, blind call transfer. When a call transfer between two lines is initiated while the destination line is not yet connected, this is a blind call transfer. The transfer source line will stay in state transferring while the transfer is still in progress. The peer on that line will hear an alerting sound.
- **Disabled**: this is a special inactive state. Temporarily no calls are accepted on this line.
- **DirectCall**: incoming intercom call, the connection is established but the micro is muted. The peer can talk to us.



The following diagram displays a “simple” view of the line state machine. The diagram is not complete but should give an idea how a SwyxIt! line works.



1.6 CClientLineMgr Methods

The CClientLineMgr object provides methods for registration, line selection and media streaming. Please note that the parameter types are depending on the used programming language and COM wrapper technique.

long DispInit(string ServerName);
Initialize line manager and connect to PBX
the method will return a HRESULT error code we get from connecting to PBX
returns S_OK when succeeded

All clients that register, have to usethe following sequence:
On startup: DispInit - DispRegisterUser (if DispInit succeeded)
On closing: DispReleaseUser (if DispRegisterUser succeeded)
If this is not done properly, the client line manger will not shut down and the user is still logged on to the PBX.

Calling DispInit, DispRegisterUser and DispReleaseUser is not required for all clients. Registration should be done only by clients that intend to provide a complete phone user interface like SwyxIt! itself and will replace SwyxIt!. For "simple" dialer applications (software that occasionally establishs a phone call) or monitor applications (software that traces incoming or calls and responds with pop up windows etc.) must not really logon (it is not recommended). All methods can be used without logon. The line manager implements a reference counter for all logged on client applications. The client line-manger unregisters from the SwyxServer only after all client applications have logged off again. Please refer to the samples: the trace tool and the dialer VB script do not register.

long DispInitEx(string ServerName, string BackupServerName);
Initialize line manager and connect to PBX
Returns HRESULT error code we get from connecting to PBX
return==S_OK -> success
All clients that register, have to use the following sequence:
On startup: DispInit - DispRegisterUser (if DispInit succeeded)
On closing: DispReleaseUser (if DispRegisterUser succeeded)
If this is not done properly, the client line manger will not shut down and the user is still logged on to the PBX. Calling DispInit, DispRegisterUser and DispReleaseUser is not required for all clients. Registration should be done only by full clients, clients that intend to provide a complete phone user interface like SwyxIt! it self.
For "simple" dialer applications (software that occasionally establishes a phone call) or monitor applications (software that traces incoming or all calls and responds with pop up windows etc.) must not really logon. All methods can be used without logon. The client line manager implements a reference counter for all logged on client applications. The client line manger un-registers from the SwyxServer only after all logged on client applications have logged off again. Please refer to the sample applications: the trace tool and the dialer VB script do not register. If you want to use server as retrieved via auto detection, please call DispInit / DispInitEx without server name

long DispRegisterUser(string UserName);
Register a user on the PBX; user name is PBX user name, not windows network username
Authentication will be done based on logged in interactive windows user
Returns unique user id; we need that user id for release user so please store it.

All clients that register have to use the following sequence:

On startup: DispInit - DispRegisterUser (if DispInit succeeded)
On closing: DispReleaseUser (if DispRegisterUser succeeded)
If this is not done properly, the client line manger will not shut down and the user is still logged on to the PBX.

Calling DispInit, DispRegisterUser and DispReleaseUser is not required for all clients. Registration should be done only by full clients, clients that intend to provide a complete phone user interface like SwyxIt! itself. For "simple" dialer applications (software that occasionally establishs a phone call) or monitor applications (software that traces incoming or all calls and responds with pop up windows etc.) must not really logon. All methods can be used without logon. The client line manager implements a reference counter for all logged on client applications. The client linemanger unregisters from the SwyxServer only after all logged on client applications have logged off again.
Please refer to the sample applications: the trace tool and the dialer VB script do not register.

DispReleaseUser(long UserId);
Logoff user from PBX use same user id as got from DispRegisterUser
All clients that register, have to use the following sequence:
On startup: DispInit - DispRegisterUser (if DispInit succeeded)
On closing: DispReleaseUser (if DispRegisterUser succeeded)
If this is not done properly, the client line manger will not shut down and the user is still logged on to the PBX.

Calling DispInit, DispRegisterUser and DispReleaseUser is not required for all clients. Registration should be done only by full clients, clients that intend to provide a complete phone user interface like SwyxIt! it self. For "simple" dialer applications (software that occasionally establishs a phone call) or monitor applications (software that traces incoming or all calls and responds with pop up windows etc.) must not really logon. All methods can be used without logon. The client line manager implements a reference counter for all logged on client applications. The client linemanger unregisters from the SwyxServer only after all logged on client applications have logged off again. Please refer to the sample applications: the trace tool and the dialer VB script do not register.

Property bool DispAutoDetectionEnabled
Get Details about server addresses retreived via AutoDetection
TRUE if AutoDetection API available, FALSE if we could not use AutoDetection (Windows 9x, NT4)

Property bool DispAutoDetectionServerAvailable
Get Details about server addresses retreived via AutoDetection
TRUE if we retreived server addresses via AutoDetection);

Property string DispAutoDetectionPrimaryServer
Get Details about server addresses retreived via AutoDetection
primary server to be used (resolved name, ip address if name could not be resolved)
If you want to use server as retrieved via AutoDetection, please call DispInit / DispInitEx without server name

Property string DispAutoDetectionBackupServer
Get Details about server addresses retreived via AutoDetection

backup server to be used (resolved name, ip address if name could not be resolved)
If you want to use server as retrieved via AutoDetection, please call DispInit / DispInitEx without server name

Property string DispGetCurrentServer
Get Logon Details about current session
*string *psServer: server we are logged on to; if NULL, we are not initialized*

Property string DispGetCurrentUser
Get Logon Details about current session
*string *psUser: user name we used; if NULL, we are not logged on*

Property bool DispIsServerUp
check whether connection to server is up (again), after we received message "PubCLMgrServer-DownMessage"
while connection to server is down, this method will try to reconnect immediately
Client Line Manager will try reconnect in fixed time interval anyway

Property long DispNumberOfLines
long DispSetNumberOfLines(long NumberOfLines);
Get / set number of lines. If number of lines is changed, we get message "PubCLMgrNumberOfLine-sChangedMessage"
PubSetNumberOfLines will only change the number of created lines, the number of configured lines will not be changed
the method will return a HRESULT error code
Returns S_OK when succeeded

ClientLine DispGetLine(long iLineNumber);
*Get interface pointer to specified line object. If *ppIClientLinePub==NULL the line is not available*

Property long DispSelectedLineNumber
long DispSelectLineNumber(long iLineNumber);
get / set selected line by index (first line == 0, ...)
the previous selected line will be set to hold / disconnected if it was active or dialing, alerting etc.
the afterwards selected line will not be activated or hooked off
the method will return a HRESULT error code
returns S_OK when succeeded
returns S_FALSE when it could not select line (probably because selected line is active and cannot be hold for some reason)

Property ClientLine DispSelectedLine
long DispSelectLine(ClientLine);
get / set selected line by interface pointer
the previous selected line will be set to hold / disconnected if it was active or dialing, alerting etc. the afterwards selected line will not be activated or hooked off
the method will return a HRESULT error code

returns S_OK when succeeded
returns S_FALSE -> could not select line (probably because selected line is active and cannot be hold for some reason)

long DispSwitchToLineNumber(long iLineNumber);
switch to line, the previous selected line will be set to hold / disconnected if it was active or dialing, alerting etc. after a short timeout the afterwards selected line will -> be activated if currently on hold
-> pickup incoming call if ringing
-> hook off if currently inactive / terminated
the method will return a HRESULT error code
returns S_OK -> success
returns S_FALSE -> could not select line (probably because selected line is active and cannot be hold for some reason)

long DispSwitchToLine(ClientLine);
switch to line, the previous selected line will be set to hold / disconnected if it was active or dialing, alerting etc. After a short timeout the afterwards selected line will -> be activated if currently on hold
-> pickup incoming call if ringing
-> hook off if currently inactive / terminated
the method will return a HRESULT error code
returns S_OK -> success
returns S_FALSE -> could not select line (probably because selected line is active and cannot be hold for some reason)

Property bool DispMicroEnabled
DispMicroEnabled(bool bEnabled);
get / set microphone state

Property bool DispSpeakerEnabled
DispSpeakerEnabled(bool bEnabled);
get / set speaker state

DispRegisterMessageTarget(long hWnd, long dwThreadId);
register thread / window for window messages
client line manager will signal events to us by sending windows messages to a given window or thread
in order to receive events you need to capture the windows message id as returned by RegisterWindowMessage(_T("WM_LineMgrNotification"));
in order to have the messages send to a window, register the window handle
in order to have the messages send to a certain thread, register its thread id

DispUnRegisterMessageTarget(long hWnd, long dwThreadId);
unregister thread / window for window messages

DispPickupGroupNotificationCall(long LineNumber);
pickup group call on specified line
set line = -1 if line does not matter -> select first inactive line

Property string DispNotificationCallPeerNumber
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
number of caller

Property string DispNotificationCallPeerName
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
name of caller

Property string DispNotificationCallCalledExtension
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
one of our own extensions or number of called group (it it was group call)

Property string DispNotificationCallCalledName
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
name of called PBX user = our own name or name of group that has been called

Property string DispNotificationCallRedirectedFromNumber
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
number of PBX user that has forwarded this call

Property string DispNotificationCallRedirectedFromName
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
name of PBX user that has forwarded this call

Property bool DispNotificationCallWasRedirected
available notification call, signaled with "PubCLMgrGroupCallNotificationMessage"
we can pick up call for other user by "DispPickupGroupNotificationCall"
this call was redirected (incoming or outgoing)

Property string DispGetCallbackOnBusyNotifyPeerName
get details of current callback on busy notification: peer name that is now free again

Property string DispGetCallbackOnBusyNotifyPeerNumber
get details of current callback on busy notification: peer name that is now free again

DispPickupCallbackOnBusyNotification(int LineNumber);

pickup callback on busy -> retry calling on specified line
line = -1 if line does not matter -> select first inactive line

DispRejectCallbackOnBusyNotification();
reject callback on busy -> do not retry calling, stop notification

DispRequestCallbackOnBusy(string PeerNumber, string PeerName);
request callback on busy: peer is busy, server will notify us as soon as peer is no more busy
peer will not get any info about RequestCallbackOnBusy
The peer number is important here; the name will be used for display during callback signaling

Property long DispNumberOfExtensions
Get number of configured extensions

string DispGetExtension(long iIndex);
Get configured extension with index i

Property long DispNumberOfSpeedDials
Get number of configured speed dials

string DispSpeedDialName(long iIndex);
Get speed dial configuration: Name

string DispSpeedDialNumber(long iIndex);
Get speed dial configuration: Dialnumber

long DispSpeedDialState(long iIndex);
Get speed dial notification state
speed dial signalling states
PubSDUnknown=0, //unknown -> external number
PubSDLoggedOut=1, //known user, not logged on to PBX
PubSDLoggedIn=2, //known user, logged on to PBX
PubSDBusy=3, //known user, client is busy
PubSDGroupCallNotification=4 //known user, that user currently receives a call; notification call

DispSimpleDial(string dialstring);
dial given dialstring on any free line
If selected line is active, an inactive line will be selected automatically before dialing
function call will be handled asynchronous, return code S_OK does not guarantee success

long DispSimpleDialEx(string dialstring);
dial given dialstring on any free line
If selected line is active, an inactive line will be selected automatically before dialing

function call will be handled asynchronous, return code S_OK does not guarantee success
*If *pdwErrorCode==S_OK, the number was dialed asynchronous.*
*If *pdwErrorCode==S_FALSE, the number was not dialed. The phone client software is not running (line manager not logged on to server)*
Any other errorcode: internal errocode, might come from server or line manager

long DispSimpleDialEx2(string dialstring, int LineNumber);
LineNumber=0..n: dial given dialstring on this line if free line
LineNumber=-1: dial given dialstring on any free line
If chosen line or selected line is active, an inactive line will be selected automatically before dialing
If selected line is active, an inactive line will be selected automatically before dialing
function call will be handled asynchronous, return code S_OK does not guarantee success
*If *pdwErrorCode==S_OK, the number was dialed asynchronous.*
*If *pdwErrorCode==S_FALSE, the number was not dialed. The phone client software is not running (line manager not logged on to server)*
Any other errorcode: internal errocode, might come from server or line manager

long DispSimpleDialEx3(string dialstring, int LineNumber, bool bProcessNumber, string name);
LineNumber=0..n: dial given dialstring on this line if free line
LineNumber=-1: dial given dialstring on any free line
If chosen line or selected line is active, an inactive line will be selected automatically before dialing
If selected line is active, an inactive line will be selected automatically before dialing
function call will be handled asynchronous, return code S_OK does not guarantee success
bProcessNumber: TRUE: automatically add public access code if it is possibly missing
name: if known, the name might be submitted as well; will be used for name resolution
*If *pdwErrorCode==S_OK, the number was dialed asynchronous.*
*If *pdwErrorCode==S_FALSE, the number was not dialed. The phone client software is not running (line manager not logged on to server)*
Any other errorcode: internal errocode, might come from server or line manager

DispVoicemailRemoteInquiry();
Dial own number for entering voicemail remote inquiry menu

string DispResolveNumber(string Number);
This method will resolve a phone number to a name
Input: Number
Output: Name.
*If the number can be resolved, *pName is the resolved name and the HRESULT is S_OK.*
*If the number cannot be resolved, *pName is set to NULL and the HRESULT is S_FALSE.*
If Number is NULL or pName is NULL, HRESULT will be E_INVALIDARG

string DispConvertNumber(DWORD Style, string NumberFrom);
This method will convert a phone number to a different format
requested style:
PubCLMgrNumberStyleFull=0, //e.g. 0004923147770 or 0023147770
PubCLMgrNumberStylePlain=1, //e.g. 4923147770 or 23147770
PubCLMgrNumberStyleCanonical=2 //e.g. +49 (231) 47770 or +49 (231) 4777-0

Property string DispCountryCode
Get CountryCode
pCountryCode: "49" for Germany, without leading zeros

Property string DispAreaCode
Get AreaCode
pAreaCode: "231" for Dortmund, without leading zeros

Property string DispPublicAccessPrefix
Get PublicAccessPrefix
pPublicAccessPrefix: e.g. "0" -> a user has to dial "0" for a public line

Property string DispLongDistanceCallPrefix
Get LongDistanceCallPrefix
pLongDistanceCallPrefix: e.g. "0" -> a user has to dial "00" (public 0 + long distance 0) for long distance calls

Property string DispInternationCallPrefix
Get InternationCallPrefix
pInternationCallPrefix: e.g. "00" -> a user has to dial "000" (public 0 + international 00) for international calls

DispCreateConference(long iConferenceLine);
Create new conference
If iConferenceLine == -1, active line or first line on hold will become conference line.
Otherwise iConferenceLine indicicates the line that will become the conference line
The chosen line must be active or on hold.

DispJoinAllToConference(bool bCreateConference);
Add all active and onhold line to a conference.
If bCreateConference==TRUE, a new conference will be created if there is currently no conference

Property bool DispConferenceRunning
Conference established or in creation?

Property ClientLine DispConferenceLine
Get conference line; returns NULL if no conference running

Property long DispConferenceLineNumber
Get zero based index of conference line; returns -1 if no conference running

DispJoinLineToConference(long iLine);
Add requested line (zero based index) to running conference

DispPlaySoundFile(string sFileName, bool bPlayLoop, int iLoopInterval);
Play soundfile using our mediastreaming
The file will be heared locally and remote (if in a phone call and voice connection is established, so not on hold)
sFileName: full path of the file to be played
bPlayLoop: if TRUE, the file will be played repeatedly
iLoopInterval: time interval (Seconds) between loops; ignored if bPlayLoop==FALSE
Works only in normal operation mode

DispStopPlaySoundFile();
Stop playback of currently played sound file
Works only in normal operation mode

DispPlayToRtp(string sFullPath, bool bLoop, DWORD dwPause);
Play soundfile using our mediastreaming
The file will be heared only remote (if in a phone call and voice connection is established, so not on hold)
The function may be called any time, it is not related to a certain connection or line.
If lines are switched, the playback will proceed.
sFullPath: Full file name
bLoop: if TRUE, the file is played in loop, otherwise it is played once
dwPause: if playing in loop, this parameter gives the pause beween loops, milliseconds
Works only in normal operation mode

DispStopPlayToRtp();
Stop playing soundfile towards RTP

DispRecordFromRtp(string sFullPath, bool bAppend, bool bAddLocalSounds);
Record the incoming voice (RTP) stream of an active connection.
The local (own) voice stream is not recorded. Only local sounds (dial tone, alerting, etc.) can be recorded as well.
The function may be called any time, it is not related to a certain connection or line.
If lines are switched, the recording will proceed.
sFullPath: Full file name
bAppend: if TRUE, the file is appended; otherwise the file is overwritten if it exists
bAddLocalSounds: if TRUE, local notification sounds are recorded too; otherwise only the RTP stream is recorded
Works only in normal operation mode

DispStopRecordFromRtp();
Stop recording RTP stream to soundfile
Works only in normal operation mode

string DispCreateMediastreamingLink();
Power dial mode only
Create a media streaming link object for linking the RTP connections of multiple lines in power dial mode
Returns an GUID as reference for the created object. This GUID can passed to multiple lines for connecting the media streaming

DispDeleteMediastreamingLink(string MsLinkId);
Power dial mode only
Delete a media streaming link object given by its GUID. The GUID should not passed to lines afterwards.

1.7 CClientLine Methods

The CClientLine object provides methods for call handling.

DispHookOff();
hook up before dialing or pick up incoming call
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispHookOffEx(string CallerId, bool SuppressRedialListEntry);
hook up before dialing or pick up incoming call
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state
CallerId is the own number (and therefore SIP account) to be used for this call
If SuppressRedialListEntry is set to true, the next call will not be shown in the redial list

DispHookOn();
hook on, disconnect call, reject incoming call
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispPressHook();
Press hook (hook on, and if handset is offhook, hook off again)
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispDial(string dialstring);
dial given dialstring, or send that string as DTMF signal (if connected)
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we get message "PubCLMgrLineDetailsChangedMessage" and property "DispAcknowledgedDialstring" will contain dialstring that has been dialed so far

and property "DispState" will contain new state

DispHold();
set an active line on hold, does not activate an other line
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispActivate();
reactive a line that is currently on hold
line is selected automatically, an other active line will be set to hold
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispDirectCall();
line is alerting: connect in direct call mode
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispJoinConference(ClientLine conference line);
Add this line to a conference / connection on given line
(pConferenceLine). If connection on pConferenceLine was not yet
a conference, it will be transferred into a conference. This
line will be disconnected afterwards.
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispTransferCall(ClientLine pITargetLine);
Transfer the connection from this line to line given in
pTargetLine. Both lines will be disconnected on this client,
and both peers will be connected afterwards.
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispForwardCall(string dialstring);
Forward an established call to a specified dialnumber
function call will be handled asynchronous, return code S_OK does not guarantee success
if successful, we will get message "PubCLMgrLineStateChangedMessage"
and property "DispState" will contain new state

DispEnable(bool bEnable);
enable / disable line for incoming calls

(manual wrap up time; disabling does not start timer for automatical enabling line)

DispRequestCallbackAuto();
request callback depending on line state
if destination is busy or if it is a second call, we will do a callback on busy
in all other cases we will request a callback

DispStartRecording();
record current conversation
file name will be build of own extension, peer extension / name and timestamp
location depends on configuration
depending on configuration, inquiry calls will be recorded into same file or different file
Works only in normal operation mode

DispStopRecording();
stop recording of current conversation

DispSetMediastreamingLink(string MsLinkId);
Power dial mode, link the media streaming of this line to media streaming link object given by its GUID
Reset the linkage by setting MsLinkId to NULL or ""

DispPlaySoundFile(string sFileName, bool bPlayLoop, int iLoopInterval);
Play soundfile using our mediastreaming
Power dial mode; Sound will be heared by all lines connected to the same media streaming link

DispStopPlaySoundFile();
Stop Play soundfile

DispRecordSoundFile(string sFileName, bool bAppend);
Record soundfile using our mediastreaming
Power dial mode; Record all sound from all lines connected to the same media streaming link

DispStopRecordSoundFile();
Stop Record soundfile

Property string DispGetIncomingExtension
Get configured extension for incoming calls on this line
returns empty string if we accept calls for any extension

Property string DispGetOutgoingExtension
Get configured extension used for outgoing calls on this line

Property long DispState
get state of that line
we will notified about changed state with message "PubCLMgrLineStateChangedMessage"
possible states are:

PubLSInactive=0, //line is inactive
PubLSHookOffInternal=1, //off hook, internal dialtone
PubLSHookOffExternal=2, //off hook, external dialtone
PubLSRinging=3, //incoming call, ringing
PubLSDialing=4, //outgoing call, we are dialing, no sound
PubLSAlerting=5, //outgoing call, alerting = ringing on destination
PubLSKnocking=6, //outgoing call, knocking = second call ringing on destination
PubLSBusy=7, //outgoing call, destination is busy, reason given in property "DispDiscReason"
PubLSActive=8, //incoming / outgoing call, logical and physical connection is established
PubLSOnHold=9, //incoming / outgoing call, logical connection is established, destination gets music on hold
PubLSConferenceActive=10, //incoming / outgoing conference, logical and physical connection is established
PubLSConferenceOnHold=11, //incoming / outgoing conference, logical connection is established, not physcically connected
PubLSTerminated=12 //incoming / outgoing connection / call has been disconnected,
//reason given in property "DispDiscReason"
PubLSDirectCall=13 //incoming call, logical and physical connection is established but micro is muted

Property string DispAcknowledgedDialstring
dialstring that has been acknowledged by server so far
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispPeerNumber
phone number of peer, incoming and outgoing
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispPeerName
name of peer, incoming and outgoing
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispCalledExtension
extension of called line (incoming call: one of our own extensions, or in case of a group call the group extension)
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispCalledName
name of called PBX user (incoming call: our name, or in case of a group call the group name)
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property bool DispIsDirectCall
this call ought to be a intercom call
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"
not implemented in SwyxWare 2.0, will always be FALSE

Property bool DispCallWasRedirected
this call was redirected (incoming and outgoing)
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property bool DispIsGroupCall
this call was a directed to the whole group
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispRedirectedFromNumber
number of PBX user that forwarded this call
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispRedirectedFromName
name of PBX user that forwarded this call
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property string DispChargingDetails
string containing charging information
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property long DispDiscReason
disconnect reason
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"
reason for disconnect on line states "PubLSBusy" and "PubLSTerminated"
possible reasons:
PubSCl3DRNormal=0, //nothing special
PubSCl3DRBusy=1, //peer was busy
PubSCl3DRRejected=2, //peer rejected call
PubSCl3DRCancelled=3, //call was cancelled
PubSCl3DRTransferred=4, //call was transferred (connect to peers with "PubTransferCall")
PubSCl3DRJoinedConference=5, //call was added to conference on other line with "PubJoinConference"
PubSCl3DRNoAnswer=6, //peer did not pick up, timeout
PubSCl3DRTooLate=7, //call was already picked up from other phone
PubSCl3DRDirectCallImpossible=8, //direct call to peer was not allowed
PubSCl3DRWrongNumber=9, //invalid number was dialed
PubSCl3DRUnreachable=10, //destination is unreachable
PubSCl3DRCallDiverted=11, //call was redirected
PubSCl3DRCallRoutingFailed=12, //call routing failed, possible script error (script of peer)
PubSCl3DRPermissionDenied=13, //permission for call was denied due to call restrictions (e.g. no long distance call allowed)
PubSCl3DRNetworkCongestion=14, //no line available (public ISDN network)

PubSCl3DRNoChannelAvailable=15, //no gateway channel available
PubSCl3DRNumberChanged=16, //number of destination has changed
PubSCl3DRIncompatibleDestination=17, //destination is incompatible (compression)

Property System.DateTime DispConnectionStartTime
start time of current connection
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property System.DateTime DispConnectionFinishedTime
end time of current connection
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"

Property bool DispIsOutgoingCall
is current call incoming or outgoing?

Property long DispCallId
Unique call id (valid only for incoming calls); will be 0 for outgoing calls

Property string DispReceivedDtmfDigits
DTMF digits received from peer

Property bool DispLastCallIsOutgoingCall
was last call incoming or outgoing?

Property long DispLastCallDiscReason
disconnect reason
we will notified about changed data with message "PubCLMgrLineDetailsChangedMessage"
reason for disconnect on line states "PubLSBusy" and "PubLSTerminated"

Property bool DispLastCallWasConnected
was last call connected?

Property long DispLastCallId
Unique call id of last call (valid only for incoming calls); will be 0 for outgoing calls

Property string DispLastCallExtension
our extension that was used for last call
we will notified about changed data with message "PubCLMgrCallDetailsMessage"

Property string DispLastCallChargingDetails
charging of last call
we will notified about changed data with message "PubCLMgrCallDetailsMessage"

Property string DispLastCallPeersPbxUserName
peer name of last call
we will notified about changed data with message "PubCLMgrCallDetailsMessage"

Property string DispLastCallPeersNumber
peer number of last call
we will notified about changed data with message "PubCLMgrCallDetailsMessage"

Property System.DateTime DispLastCallStartTime
Start time of last call. We will be notified about changed data with message "PubCLMgrCallDetails-Message"

Property System.DateTime DispLastCallFinishedTime
End time of last call. We will be notified about changed data with message "PubCLMgrCallDetails-Message"

1.8 Structures

The reference for all supported structures is given in the included files “CLMgrPub.idl” and “CLMgrPubTypes.h”.

1.9 Enumerations

The reference for all supported enumerations (line states, line events, disconnect reasons…) is given in the included files “CLMgrPub.idl” and “CLMgrPubTypes.h”.

1.10 CClientLineMgr Events

The following events are supported by the line manager;

PubCLMgrLineStateChangedMessage=0
State of a line has changed, lParam: index of line

PubCLMgrLineSelectionChangedMessage=1
Line selection has changed, new line in focus, lParam: index of now selected line

PubCLMgrLineDetailsChangedMessage=2
Details of line have changed, lParam: index of line

PubCLMgrUserDataChangedMessage=3
User settings have been changed.

PubCLMgrCallDetailsMessage=4
Details of last call are available, after disconnect, for logging purpose. lParam: index of line

PubCLMgrServerDownMessage=5
Server is down, will be rebooted... -> keep interfaces to line manager, wait for ServerUp message

PubCLMgrServerUpMessage=6
Server is up again -> keep interfaces to line manger

PubCLMgrGroupCallNotificationMessage=8
Notification about signaled call we could pick up now
lParam: 1:signaled call available
lParam: 0:signaled call not available / no longer available

PubCLMgrNameKeyStateChangedMessage=9
notification about changed namekey state
lParam: index of namekey that has new state

PubCLMgrNumberOfLinesChangedMessage=10
the number of lines has changed
lParam: new number of lines

PubCLMgrClientShutDownRequest=11
Client Line Manager requests client to shutdown and release all interfaces

PubCLMgrPowerSuspendMessage=12
client machine goes to hibernate or suspend mode -> keep interfaces to line manager, wait for Power-Resume message

PubCLMgrPowerResumeMessage=13
client machine has returned from suspend mode -> keep interfaces to line manager

PubCLMgrCallbackOnBusyNotification=18
informing clients about available callback on busy
lParam=1: callback available; lParam=0: callback no longer available
lParam=2: callback back on busy was signaled but no user response (pickup / reject)
lParam=3: there is no "signaled callback on busy" left -> reset button state

PubCLMgrCtiPairingStateChanged=20
informing clients about changed state of CTI pairing process
lParam=new CTI pairing state
CtiSessIdle=0, no cti activity
CtiSessMasterPairingRequested=1, CTI master has requested pairing
CtiSessSlavePairingRequested=2, CTI slave has received pairing request
CtiSessMasterPaired=3, CTI master has established pairing
CtiSessSlavePaired=4, CTI slave has established pairing

PubCLMgrLineStateChangedMessageEx=28
state of at least line has changed
lParam: LOBYTE: line index, HIBYTE: new state

PubCLMgrSIPRegistrationStateChanged=30
Registration state of SIP account has changed
lParam: LOBYTE: Account index, HIBYTE: new state

PubCLMgrWaveFilePlayed=31
wave file playback finished
lParam: line index;

if -1, the message is related to a LineMgr function PlaySoundFile or PlayToRtp
if >=0 the message is related to a line function PlaySoundFile of line with this index

2 Programming Guide

2.1 Creating Line Manager Object and Accessing Lines

2.1.1 VB Script

The following code show the instantiation of a line manager object and calling some methods from VB script. When using VB 6 or VB script, you have to use the Dispatch methods (all methods where the name starts with Disp, like DispSimpleDial, DispHookOff, DispHookOn...).

```
Dim PhoneLineMgr : Set PhoneLineMgr = Nothing
Dim PhoneLineFocus : Set PhoneLineFocus = Nothing
```

Create a line manager instance:

```
Set PhoneLineMgr = Wscript.CreateObject("CLMgr.ClientLineMgr")
```

Dial a number:

```
PhoneLineMgr.DispSimpleDial("001191")
```

Get the selected line:

```
Set PhoneLineFocus = PhoneLineMgr.DispSelectedLine
```

Hook on the selected line. This will terminate the call.

```
PhoneLineFocus.DispHookOn()
```

Clean up:

```
Set PhoneLineMgr = Nothing
Set PhoneLineFocus = Nothing
```

2.1.2 VB 6

With Visual Basic 6 the Client SDK can be added to a project using menu Project\References. Check the library "CLMgr 2.0 Type Library". Afterwards the Object Browser will list all available objects and methods for library CLMGRLib. Due to technical reasons you will see many more objects and interfaces than you can use. From Visual Basic you can use the objects CClientLineMgr and CClientLine and its methods and properties starting with Disp (like DispHookOff). Only the object CClientLineMgr will be created by your application. CClientLine objects are not created directly, but you will access the line by calling appropriate CClientLineMgr functions. The following example retrieves the selected line and puts it on hold:

```
Public WithEvents PhoneLineMgr As CLMGRLib.ClientLineMgr
Set PhoneLineMgr = CreateObject("CLMgr.ClientLineMgr")
Public PhoneLineFocus As Object
Set PhoneLineFocus = PhoneLineMgr.DispSelectedLine
PhoneLineFocus.DispHold
```

For receiving line manager events with Visual Basic define a CClientLineMgr object:
Public WithEvents PhoneLineMgr As CLMGRLib.ClientLineMgr

Create an object:
Set PhoneLineMgr = CreateObject("CLMgr.ClientLineMgr")

Add the following function to your code that will receive the events. The meaning of parameter param is explained in file "CLMgrPubTypes.h", included in the SDK.

```
Sub PhoneLineMgr_DispOnLineMgrNotification(ByVal msg As Long, ByVal
param As Long)
 Select Case msg
  Case 0 'PubCLMgrLineStateChangedMessage
  Case 1 'PubCLMgrLineSelectionChangedMessage
  Case 2 'PubCLMgrLineDetailsChangedMessage and so on
End Select
End Sub
```

2.1.3 VB .Net (Visual Studio 2005)

Add a reference to COM component "CLMgr 2.0 Type Library".

Import the library:

```
Imports CLMGRLib
```

Declare and create a line manager object:

```
Private WithEvents MyClmgr As New ClientLineMgrClass
```

Declare an event handler that receives line manager events from that MyClmgr object. This method will be called when the line manager sends a message to your application:

```
Private Sub clmgr_PubOnLineMgrNotification(ByVal msg As Integer, ByVal
param As Integer) Handles MyClmgr.PubOnLineMgrNotification
```

The parameter msg will be the line manager event (see enum PubCLMgrMessages in file "CLMgrPubTypes.h", included in the SDK). The parameter param depends on the message; please refer to the comments within "CLMgrPubTypes.h".

Within your form load method you have to call:

```
Control.CheckForIllegalCrossThreadCalls = False
```

This prevents thread exceptions which would occur otherwise when accessing the form from the event handler because the line manager events will come in from a different thread.

Get the selected line and do something with it:

```
Private SelectedLine As ClientLine
Dim i As Integer
SelectedLine = MyClmgr.DispSelectedLine
i=SelectedLine.DispState
SelectedLine.DispHookOff()
SelectedLine.DispHookOn()
```

2.1.4 C# .Net (Visual Studio 2005)

This is similar to VB .Net. Please refer to the provided sample code.

Add a reference to COM component "CLMgr 2.0 Type Library".

Import the library:

```
using CLMGRLib;
```

Declare and create a line manager object:

```
private ClientLineMgrClass MyCLMgr = new ClientLineMgrClass();
```

Declare an event handler that receives line manager events from that MyClmgr object. This method will be called when the line manager sends a message to your application:

```
private void clmgr_PubOnLineMgrNotification(int msg, int param)
```

The parameter msg will be the line manager event (see enum PubCLMgrMessages in file "CLMgrPubTypes.h", included in the SDK). The parameter param depends on the message; please refer to the comments within "CLMgrPubTypes.h".

The event handler has to be registered with the MyCLMgr object:

```
IClientLineMgrEventsPub_PubOnLineMgrNotificationEventHandler myHandler =
                new IClientLineMgrEventsPub_PubOnLineMgrNotificationEventHandler(
                                                    clmgr_PubOnLineMgrNotification);
MyCLMgr.PubOnLineMgrNotification += myHandler;
```

Within your form load method you have to call:

```
Control.CheckForIllegalCrossThreadCalls = false;
```

This prevents thread exceptions which would occur otherwise when accessing the form from the event handler because the line manager events will come in from a different thread.

Get the selected line and do something with it:

```
private ClientLine SelectedLine;
int i;
SelectedLine = (ClientLine)MyCLMgr.DispSelectedLine;
i=SelectedLine.DispState;
SelectedLine.DispHookOff();
SelectedLine.DispHookOn();
```

2.1.5 C++

For Visual C++ add the files CLMgrPubTypes.h, CLMgrPubTypes.c und CLMgrPub.idl to your project. You will create a Client Line Manager object (CLSID_ClientLineMgr) and use its interfaces like IClientLineMgrPub or IClientLineMgrPub2. The Client Line Manager object provides methods for accessing lines. Using the line interface IClientLinePub you may initiate actions on a line or retrieve line details. This will be shown in the first example "Visual C++ Simple". Please refer to the provided sample code. Please pay attention that you can receive line manager events as either window messages or COM events. Most C++ samples will show both ways. There are different build targets for using window messages or COM events. The related event handling code is enclosed in #ifdef blocks. The advantage of window messages is that you will receive the events within your main message pump. SwyxIt! is using that method as well. Receiving COM events in C++ is a little bit more complicated. The sample code shows how to do this using the Active Template Library, connection points and event sinks.

2.2 Linking Media Streaming of Lines in Power Dial Mode

The following example in VB script shows calling out on two lines and linking the media streaming of the two lines. Then you have to wait till both lines are disconnected later on:

```
Option Explicit
```

```
Dim PhoneLineMgr : Set PhoneLineMgr = Nothing
Dim PhoneLine1 : Set PhoneLine1 = Nothing
Dim PhoneLine2 : Set PhoneLine2 = Nothing
Dim errval, DialDigits, Pos, i
Dim sLink

Set PhoneLineMgr = Wscript.CreateObject("CLMgr.ClientLineMgr") : CheckError
Set PhoneLine1 = PhoneLineMgr.DispGetLine(0)
Set PhoneLine2 = PhoneLineMgr.DispGetLine(1)

sLink=PhoneLineMgr.DispCreateMediastreamingLink()

PhoneLine1.DispSetMediastreamingLink(sLink)
PhoneLine2.DispSetMediastreamingLink(sLink)

PhoneLine1.DispHookOff()
PhoneLine1.DispDial("12345")
PhoneLine2.DispHookOff()
PhoneLine2.DispDial("67890")

Wscript.Sleep 1000

While( (PhoneLine1.DispState>0) And (PhoneLine2.DispState>0) )
Wscript.Sleep 1000
Wend

PhoneLine1.DispSetMediastreamingLink("")
PhoneLine2.DispSetMediastreamingLink("")
PhoneLineMgr.DispDeleteMediastreamingLink(sLink)


Set PhoneLineMgr = Nothing
Set PhoneLine1 = Nothing
Set PhoneLine2 = Nothing

Sub CheckError
Dim message, errRec
If Err = 0 Then Exit Sub
message = Err.Source & " " & Hex(Err) & ": " & Err.Description
Fail message
End Sub

Sub Fail(message)
Wscript.Echo message
Wscript.Quit 2
End Sub
```

So when you are using power dial mode, the lines are fully operational, only that the media streaming is not linked at all to the local sound device. But when you connect the media streaming of two lines (using SwyxIt! 6.02 or later), the line manager will link the media streaming of the two participants. When line 1 gets early tones, the peer on line 2 will hear it. When line 2 is connected to a script and gets any music on hold and line 1 is connected to the external party, the external party will hear the music on hold.
The media streaming is then routed through the line manager.
The drawback is: you need to keep the two lines active on the power dial client. You could initiate a transfer later on (when you are sure that the external party is connected to the agent) but this would result in a short interruption of the call. So best practice would be to wait till both lines are disconnected (as shown in my sample code).
My favorite view of this: It is like "Fräulein vom Amt" sitting in front of a huge switchboard. You can have multiple calls on the lines and patch the connections independently from the call states.

There is no build in maximum limit of lines in power dial mode. So you can call DispSetNumberOf-Lines(500) and have 500 lines. But have in mind that you then might have 500 active connections with 250 media streaming links. You have to test what is possible with a given hardware.

You can link the media streaming of more than two lines as well. For more information please refer to sample “Visual Studio.Net 2005 C# PowerDialTest”.

2.3 Sample Code

The SDK contains several sample projects. You will need Visual Studio 2005 for opening the project files. Please download the Client SDK 4.x if you need project files for VB6 or Visual Studio 2003.

2.3.1 Visual Basic Script

This sample just shows how to dial a phone number via script.

2.3.2 Visual C++ ATL PlugIn

This sample explains the implementation of a Line Manager Plugin providing name resolution for SwyxIt! The code is based on the Active Template Library.
It implements a Client Line Manager PlugIn that provides name resolution for SwyxIt!. For unknown peer numbers the Client Line Manager will ask all installed PlugIns for a matching name resolution. So you will be able to provide name resolution based on your own database application. This client too will not register with the SwyxServer.
For a customization one has to replace the GUIDs and friendly class names PlugInSample.MyResolver etc. in the files PlugInSample.idl and MyResolver.rgs with new GUIDs and names in order to avoid conflicts with other 3rd party applications based on the Client SDK. But never ever change any GUID from the files CLMgrPub.idl and CLMgrPubTypes.c!
In file PlugInSample.cpp the functions for registration and deregistration of the PlugIn are implemented. In order to be loaded by the Client Line Manager, the PlugIn writes its class ID into the Line Managers registry HKLM\SOFTWARE\Swyx\Client Line Manager\CurrentVersion\Options\PlugIns\{GUID}
The actual PlugIn object CMyResolver is defined and declared in the files MyResolver.cpp and MyResolver.h. The PlugIn object implements the interfaces IClientAddInLoader, IClientResolverAddIn and IClientLineMgrEventsDisp. The interface IClientAddInLoader will be used by the Line Manager on loading and releasing the PlugIn, the interface IClientResolverAddIn provides name resolution. The interface IClientLineMgrEventsDisp finally receives the Line Manager events.
When loading the PlugIn, the Line Manager calls the functions IClientAddInLoader::Initialize, IClientAddInLoader::GetName and IClientAddInLoader::GetVersion. Before releasing the interface IClientAddInLoader and unloading the PlugIn, the Line Manager calls the method IClientAddInLoader::UnInitialize. The PlugIn has then a chance for cleaning up memory and other resources.
In the example implementation of CMyResolver::Initialize the PlugIn registers an event sink towards the Line Manager for receiving events. The Line Manager interface pointer pIClientLineMgrPub will be stored in the global interface table. Using its cookie m_dwCLMgrCookie the pointer will be retrieved on demand from the global interface table and automatically marshalled into the current thread context. You will find the used helper functions in the files githelp.cpp and githelp.h. This effort is required as soon as a COM interface pointer is used from multiple threads. GetName are GetVersion unspectacular. In the implementation of CMyResolver::UnInitialize the event sink will be released, that's always a good idea.
Line Manager events drop in in function CMyResolver::DispOnLineMgrNotification. Your PlugIn could e.g. write journal entries into a database or trigger a database application for popping up contact information.
In CMyResolver::GetPreferredNumberStyle the PlugIn can ask for a phone number format to be used for name resolution. Usually one would use the format PubCLMgrNumberStyleFull (like

0004923147770 or 0023147770). For any number to be resolved the Client Line Manager will call CMyResolver::ResolveNumber.

2.3.3 Visual C++ Call Log

This is a simple MFC application that displays a simple call log. A checkbox allows to accept all incoming calls automatically. This client runs besides SwyxIt! and does not log on nor off. Here too there are two build targets "Win32 Debug Using Event Sink" and "Win32 Release Using Event Sink" showing both usage of connection points and COM events. The class CCLMgrEventSink may be used modified in your project.

2.3.4 Visual C++ Chat

This is a simple dialog based MFC Application for chatting between two SwyxIt! client. The “To” drop list is initially filled in with the names of the configured speed dials (assuming that these are all valid SwyWare user names). When in a call, the “To” field is initialized with the peer name.

2.3.5 Visual C++ PlayToRtp

Dialog based MFC Application, allowing the recording and playback of voice stream. As “Play File” please choose a wave file in format PCM, 8000 Hz, 16 bit, mono. The playback is started by pressing “Start”. The file can be played in a loop, the pause between loops can be configured. The playback is heard only by the peer.
This sample does not work in power dial mode.

2.3.6 Visual C++ Simple

Dialog based MFC Application that displays information about the selected line and allows simple call control. The sample includes log on and log off code. Nevertheless logon and logoff should only be used via SDK when your application intends to run stand alone (without SwyxIt! running aside). When you use this sample along with SwyxIt!, please don’t fill in server and user name, just press logon.

2.3.7 Visual C++ WakeUp

Example for establishing bulk calls using SwyxIt! in power dial mode.

The client has to be switched to power dial mode via registry:

- No implicit line selection
- If a line is selected explicitly, the previous selected line does not become disconnected on hold
- The mediastreaming is not linked to the local sound device
- No local, line related sounds are played (ringing, dialing, alerting, …)
- Hook off/on per local handset will be ignored

Enable the power dial mode using reg file "PowerDialerMode.reg". The reg file will set as well the key "CancelBlindTransferOnVoicemail" to 0 for allowing blind call transfer (forward) to a voicemail announcement. Within SwyxIt! one should disable pop up during connection for avoiding annoying pop ups.
The sample allows dialing on all available lines simultaneously. The phone call jobs are read from file JobsToDo.txt. Every line corresponds to a call to do, described by three strings, separated by ';': The first string is the number to be dialed. On connect the call will be transferred to the number given in the second string. The third string is a counter for the retries. When clicking on start in the sample application, the dialing will be initiated after a short timeout. Clicking on stop will abort the process. In a second file all successful calls will be listed. The file JobsToDo.txt lists all not yet successful call and the current number of retries.
When trying this sample on a productive server, please be aware that it is a really good stress test tool. For real scenarios one should add some Sleep() calls in order to give the server time.

2.3.8 Visual Studio.Net 2005 C# PowerDialTest

This sample is intended to run on SwyxIt! in power dial mode. Please note that it sets the line count to 50. The actual line count can be higher than the number of lines visible on the skin.
The project was used for testing the speed of switching on media streaming between two clients. So it demonstrates how to handle media streaming in power dial mode. In fact the real test scenario was: Run SwyxIt! Now in power dial mode. Configure two SIP accounts for your local SwyxServer. Dial out on behalf of account 1 to the number / URI of account 2. So our call will come in on a second line of the same client. When you then pickup the call, you can measure how long it takes to establish the voice connection by sending out a test sound on one line and record it on the other line. Please don't use this code to claim support incidents against Swyx ;-)

Outgoing Call: Please enter the dial number. In case that you are using SwyxIt! Now in power dial mode, you have to enter the caller id as well. The "caller id" is then used to select the SIP account that is used for placing the outgoing call. If checkbox "Play Sound" is checked, the sample will play a test sound CallingPartyTestsignal1000Hz10Seconds.wav toward the destination as soon as the line becomes active. This resembles a user that immediately starts speaking when the line becomes visibly active. The incoming media streaming for this call will be recorded. For this you can see in the sample code that on line a media streaming link is created and DispRecordSoundFile is called on the line. Press "Start call" for placing the outbound call.

If the SIP accounts are configured correctly (as discussed: Use SwyxIt! Now in power dial mode, two SIP accounts), the call will come in on a second line. Accept the call by pressing "Accept Call". The sample will then display the time difference between accepting the incoming call and receiving the "connect" message on the outgoing call on the first line. This is the time difference for establishing a logical connection. If "Play Sound" is checked, the application will play a test sound CalledPartyTestsignal2000Hz10Seconds.wav towards the caller. This resembles a user that immediately starts speaking after accepting a call. The incoming media streaming for this call will be recorded.

Independent from the just described test scenario, the other buttons show how to link the media streaming of multiple lines to each other, playback sound files in power dial mode or record voice in power dial mode.

When looking into the code itself you will find as well an example of handling incoming DTMF.

2.3.9 Visual Studio.Net 2005 C# Simple

Very simple application in C# showing dialing and receiving events. Nevertheless pay attention to the code comments for avoiding nasty problems in your own code.

2.3.10 Visual Studio.Net 2005 VB Simple

Very simple application in VB showing dialing and receiving events. Nevertheless pay attention to the code comments for avoiding nasty problems in your own code.

2.3.11 Visual Studio.Net C# IpPbxMPC

This example is a complete C# application named IpPBX Media Player Controller. It can be used to control a media player such as Winamp or Windows Media Player. Whenever not all SwyxIt! lines are inactive IpPBXMPC sends the configured player a "pause" command. When all SwyxIt! lines become inactive again, the player gets an "unpause" command from IpPbxMPC. Alternatively the active state of one or more speed dial keys can be used, too.

2.3.12 Visual Studio.Net C# Simple

Shows how to call Client SDK COM Api methods from a .Net Application. It's a simple Windows Forms application written in C#. To use the Client SDK COM Api in a .Net Application add the "CLMGR 2.0 Type Library" as reference to your project. Visual Studio.Net generates a runtime callable wrapper for the COM API which provides a namespace CLMGRLib which allows calling the COM API function and allows consuming events via delegates.
See function ConnectAndLogin() in Form1.cs, for an example on how to use this mechanism to receive LineManager events:

```
clmgr = new ClientLineMgrClass(); clmgr.DispOnLineMgrNotification +=
       new CLMGRLib.IClientLineMgrEventsDisp_DispOnLineMgrNotificationEventHandler
       (this.OnLineMgrNotification);
```

This creates a delegate which is added to the DispOnLineMgrNotification event of clmgr. Whenever the line manager sends an event the following function is called in the C# application:

```
public void OnLineMgrNotification(System.Int32 msg, System.Int32 param)
```

3 Reference

3.1 Client SDK Methods

The reference for all supported interfaces and its methods is given in the included file "CLMgrPub.idl". The methods and its parameters are documented within that file. Even if you are programming in VB or C# you should read this file as a reference for understanding the parameters.

The file "CLMgrPubTypes.h" contains the enumerations for all supported events, line states, and disconnect reasons.

The file "CLMgrPubTypes.c" contains defines for some Class IDs that are required for linking C++ projects.

3.2 Supported TAPI Features

The Microsoft TAPI provides an upper level interface to be used by TAPI client applications like Windows Dialer or CRM software. Telephony applications like SwyxIt! integrate into TAPI by implementing a so called TAPI Service Provider (TSP). A TSP is a sort of driver that forwards TAPI function calls to the 3rd party telephony hardware or software components.
The SwyxIt! TSP supports TAPI 2.x. Nevertheless it can be used by TAPI 3.x client applications as long as no functions are used that are not supported by TAPI 2.x. SwyxIt! supports basic call handling including transfer and conference, there is no access to media streaming.

Please note that the "SwyxIt! TAPI Service Provider" provides some configuration that is accessible using the telephony control panel applet. Especially you can configure…

- A fixed dial prefix for outgoing calls (if you need this for any TAPI application that does not dial canonical and does not use the Windows locations neither)
- Caller ID format: the format that is used for reporting phone numbers towards TAPI. Please use the format that is appropriate for your application. For DATEV e.g. you would choose plain or full. For ACT! you would choose plain.
- "Signal outbound calls": By default the TSP does only signal incoming calls to the TAPI. So the TAPI would not be aware of outbound calls that are started using SwyxIt!. If you need the visibility of all outbound calls to your TAPI application, please check this checkbox.

SwyxIt! supports the following TAPI functions:

- lineOpen
- lineClose
- lineMakeCall
- lineDial
- lineAnswer
- lineHold
- lineUnhold
- lineSwapHold
- lineSetupTransfer
- lineBlindTransfer
- lineCompleteTransfer
- lineSetupConference
- linePrepareAddToConference

- lineAddToConference
- lineRemoveFromConference
- lineDrop
- lineCloseCall
- lineGetAddressCaps
- lineGetAddressStatus
- lineGetCallInfo
- lineGetCallStatus
- lineGetDevCaps
- lineGetID
- lineGetLineDevStatus