



# **IP Office™ Platform**

Description of MTCTI-3 API 12.2 Introduced in  
Release 11.1.0.0

Version 1.1



## AVAYA SOFTWARE DEVELOPMENT KIT LICENSE AGREEMENT

REVISED: Aug 1, 2025

© 2025 AVAYA Avaya LLC. All Rights Reserved.

### Notice

While reasonable efforts have been made to ensure that the information in this document is complete and accurate at the time of printing, Avaya assumes no liability for any errors. Avaya reserves the right to make changes and corrections to the information in this document without the obligation to notify any person or organization of such changes.

### Documentation disclaimer

“Documentation” means information published by Avaya in varying mediums which may include product information, operating instructions and performance specifications that Avaya may generally make available to users of its products and Cloud Services. Documentation does not include marketing materials. Avaya shall not be responsible for any modifications, additions, or deletions to the original published version of documentation unless such modifications, additions, or deletions were performed by Avaya. End User agrees to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation, to the extent made by End User.

### Link disclaimer

Avaya is not responsible for the contents or reliability of any linked websites referenced within this site or documentation provided by Avaya. Avaya is not responsible for the accuracy of any information, statement or content provided on these sites and does not necessarily endorse the products, services, or information described or offered within them. Avaya does not guarantee that these links will work all the time and has no control over the availability of the linked pages.

### Warranty

Avaya provides a limited warranty on Avaya hardware and software. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya's standard warranty language, as well as information regarding support for this product while under warranty is available to Avaya customers and other parties through the Avaya Support website: <https://support.avaya.com/helpcenter/getGenericDetails?detailId=C20091120112456651010> under the link “Warranty & Product Lifecycle” or such successor site as designated by Avaya. Please note that if You acquired the product(s) from an authorized Avaya Channel

Partner outside of the United States and Canada, the warranty is provided to You by said Avaya Channel Partner and not by Avaya.

“Cloud Cloud Service” means a cloud service subscription that You acquire from either Avaya or an authorized Avaya Channel Partner (as applicable) and which is described further in the applicable Service Description or other service description documentation regarding the applicable cloud service. If You purchase a Cloud Service subscription, the foregoing limited warranty may not apply but You may be entitled to support services in connection with the Cloud Service as described further in your service description documents for the applicable Cloud Service. Contact Avaya or Avaya Channel Partner (as applicable) for more information.

### Cloud Service

THE FOLLOWING APPLIES IF YOU PURCHASE A CLOUD SERVICE SUBSCRIPTION FROM AVAYA OR AN AVAYA CHANNEL PARTNER (AS APPLICABLE), THE TERMS OF USE FOR CLOUD SERVICES ARE AVAILABLE ON THE AVAYA WEBSITE, <https://www.avaya.com/en/legal/license-terms/> UNDER THE LINK “Avaya Terms of Use for Cloud Services” OR SUCH SUCCESSOR SITE AS DESIGNATED BY AVAYA, AND ARE APPLICABLE TO ANYONE WHO ACCESSES OR USES THE CLOUD SERVICE. BY ACCESSING OR USING THE CLOUD SERVICE, OR AUTHORIZING OTHERS TO DO SO, YOU, ON BEHALF OF YOURSELF AND THE ENTITY FOR WHOM YOU ARE DOING SO (HEREINAFTER REFERRED TO INTERCHANGEABLY AS “YOU” AND “END USER”), AGREE TO THE TERMS OF USE. IF YOU ARE ACCEPTING THE TERMS OF USE ON BEHALF A COMPANY OR OTHER LEGAL ENTITY, YOU REPRESENT THAT YOU HAVE THE AUTHORITY TO BIND SUCH ENTITY TO THESE TERMS OF USE. IF YOU DO NOT HAVE SUCH AUTHORITY, OR IF YOU DO NOT WISH TO ACCEPT THESE TERMS OF USE, YOU MUST NOT ACCESS OR USE THE CLOUD SERVICE OR AUTHORIZE ANYONE TO ACCESS OR USE THE CLOUD SERVICE. YOUR USE OF THE CLOUD SERVICE SHALL BE LIMITED BY THE NUMBER AND TYPE OF LICENSES PURCHASED UNDER YOUR CONTRACT FOR THE CLOUD SERVICE, PROVIDED, HOWEVER, THAT FOR CERTAIN CLOUD SERVICES IF APPLICABLE, YOU MAY HAVE THE OPPORTUNITY TO USE FLEX LICENSES, WHICH WILL BE INVOICED ACCORDING TO ACTUAL USAGE ABOVE THE CONTRACT LICENSE LEVEL. CONTACT AVAYA OR AVAYA'S CHANNEL PARTNER FOR MORE INFORMATION ABOUT THE LICENSES FOR THE APPLICABLE CLOUD

SERVICE, THE AVAILABILITY OF ANY FLEX LICENSES (IF APPLICABLE), PRICING AND BILLING INFORMATION, AND OTHER IMPORTANT INFORMATION REGARDING THE CLOUD SERVICE.

## Licenses

THE SOFTWARE LICENSE TERMS AVAILABLE ON THE AVAYA WEBSITE, <https://www.avaya.com/en/legal/license-terms/>, UNDER THE LINK “AVAYA SOFTWARE LICENSE TERMS (Avaya Products)” OR SUCH SUCCESSOR SITE AS DESIGNATED BY AVAYA, ARE APPLICABLE TO ANYONE WHO DOWNLOADS, USES AND/OR INSTALLS AVAYA SOFTWARE, PURCHASED FROM AVAYA LLC, ANY AVAYA AFFILIATE, OR AN AVAYA CHANNEL PARTNER (AS APPLICABLE) UNDER A COMMERCIAL AGREEMENT WITH AVAYA OR AN AVAYA CHANNEL PARTNER. UNLESS OTHERWISE AGREED TO BY AVAYA IN WRITING, AVAYA DOES NOT EXTEND THIS LICENSE IF THE SOFTWARE WAS OBTAINED FROM ANYONE OTHER THAN AVAYA, AN AVAYA AFFILIATE OR AN AVAYA CHANNEL PARTNER; AVAYA RESERVES THE RIGHT TO TAKE LEGAL ACTION AGAINST YOU AND ANYONE ELSE USING OR SELLING THE SOFTWARE WITHOUT A LICENSE. BY INSTALLING, DOWNLOADING OR USING THE SOFTWARE, OR AUTHORIZING OTHERS TO DO SO, YOU, ON BEHALF OF YOURSELF AND THE ENTITY FOR WHOM YOU ARE INSTALLING, DOWNLOADING OR USING THE SOFTWARE (HEREINAFTER REFERRED TO INTERCHANGEABLY AS “YOU” AND “END USER”), AGREE TO THESE TERMS AND CONDITIONS AND CREATE A BINDING CONTRACT BETWEEN YOU AND AVAYA LLC OR THE APPLICABLE AVAYA AFFILIATE (“AVAYA”).

## Copyright

Except where expressly stated otherwise, no use should be made of materials on this site, the Documentation, Software, Cloud Service, or hardware provided by Avaya. All content on this site, the documentation, Cloud Service, and the product provided by Avaya including the selection, arrangement and design of the content is owned either by Avaya or its licensors and is protected by copyright and other intellectual property laws including the sui generis rights relating to the protection of databases. You may not modify, copy, reproduce, republish, upload, post, transmit or distribute in any way any content, in whole or in part, including any code and software unless expressly authorized by Avaya. Unauthorized reproduction, transmission, dissemination, storage, and or use without the express written consent of Avaya can be a criminal, as well as a civil offense under the applicable law.

## Virtualization

The following applies if the product is deployed on a virtual machine. Each product has its own ordering code and license types. Note that each Instance of a product must be separately licensed and ordered. For example, if the end user customer or Avaya Channel Partner would like to install two Instances of the same type of products, then two products of that type must be ordered.

## Third Party Components

“Third Party Components” mean certain software programs or portions thereof included in the Software or Cloud Service may contain software (including open source software) distributed under third party agreements (“Third Party Components”), which contain terms regarding the rights to use certain portions of the Software (“Third Party Terms”). As required, information regarding distributed Linux OS source code (for those products that have distributed Linux OS source code) and identifying the copyright holders of the Third Party Components and the Third Party Terms that apply is available in the products, Documentation or on Avaya’s website at: <https://www.avaya.com/en/legal/third-party-terms/> or such successor site as designated by Avaya.

The open source software license terms provided as Third Party Terms are consistent with the license rights granted in these Software License Terms, and may contain additional rights benefiting You, such as modification and distribution of the open source software. The Third Party Terms shall take precedence over these Software License Terms, solely with respect to the applicable Third Party Components to the extent that these Software License Terms impose greater restrictions on You than the applicable Third Party Terms.

The following applies if the H.264 (AVC) codec is distributed with the product. THIS PRODUCT IS LICENSED UNDER THE AVC PATENT PORTFOLIO LICENSE FOR THE PERSONAL USE OF A CONSUMER OR OTHER USES IN WHICH IT DOES NOT RECEIVE REMUNERATION TO (i) ENCODE VIDEO IN COMPLIANCE WITH THE AVC STANDARD (“AVC VIDEO”) AND/OR (ii) DECODE AVC VIDEO THAT WAS ENCODED BY A CONSUMER ENGAGED IN A PERSONAL ACTIVITY AND/OR WAS OBTAINED FROM A VIDEO PROVIDER LICENSED TO PROVIDE AVC VIDEO. NO LICENSE IS GRANTED OR SHALL BE IMPLIED FOR ANY OTHER USE. ADDITIONAL INFORMATION MAY BE OBTAINED FROM VIA LICENSING ALLIANCE. SEE <https://www.via-la.com/>.

## Service Provider

THE FOLLOWING APPLIES TO AVAYA CHANNEL PARTNER’S HOSTING OF AVAYA PRODUCTS OR SERVICES. THE PRODUCT OR CLOUD SERVICE MAY USE THIRD PARTY COMPONENTS SUBJECT TO THIRD PARTY TERMS AND REQUIRE A SERVICE PROVIDER TO BE INDEPENDENTLY LICENSED DIRECTLY FROM THE THIRD PARTY SUPPLIER. AN AVAYA CHANNEL PARTNER’S HOSTING OF AVAYA PRODUCTS MUST BE AUTHORIZED IN WRITING BY AVAYA AND IF THOSE HOSTED PRODUCTS USE OR EMBED CERTAIN THIRD PARTY SOFTWARE, INCLUDING BUT NOT LIMITED TO MICROSOFT SOFTWARE OR CODECS, THE AVAYA CHANNEL PARTNER IS REQUIRED TO INDEPENDENTLY OBTAIN ANY APPLICABLE LICENSE AGREEMENTS, AT THE AVAYA CHANNEL PARTNER’S EXPENSE, DIRECTLY FROM THE APPLICABLE THIRD PARTY SUPPLIER.

WITH RESPECT TO CODECS, IF THE AVAYA CHANNEL PARTNER IS HOSTING ANY PRODUCTS THAT USE OR EMBED THE G.729 CODEC, H.264 CODEC, OR H.265 CODEC, THE AVAYA CHANNEL PARTNER ACKNOWLEDGES AND AGREES THE AVAYA CHANNEL PARTNER IS RESPONSIBLE FOR ANY AND ALL RELATED FEES AND/OR ROYALTIES. THE G.729 CODEC IS LICENSED BY Sangoma Technologies Corporation SEE <https://www.asterisk.org/products/add-ons/g729-codec/>. THE H.264 (AVC) CODEC IS LICENSED UNDER THE AVC PATENT PORTFOLIO LICENSE FOR THE PERSONAL USE OF A CONSUMER OR OTHER USES IN WHICH IT DOES NOT RECEIVE REMUNERATION TO: (I) ENCODE VIDEO IN COMPLIANCE WITH THE AVC STANDARD (“AVC VIDEO”) AND/OR (II) DECODE AVC VIDEO THAT WAS ENCODED BY A CONSUMER ENGAGED IN A PERSONAL ACTIVITY AND/OR WAS OBTAINED FROM A VIDEO PROVIDER LICENSED TO PROVIDE AVC VIDEO. NO LICENSE IS GRANTED OR SHALL BE IMPLIED FOR ANY OTHER USE. ADDITIONAL INFORMATION FOR H.264 (AVC) AND H.265 (HEVC) CODECS MAY BE OBTAINED FROM VIA LICENSING ALLIANCE. SEE <https://www.via-la.com/>.

## **Compliance with Laws**

Customer acknowledges and agrees that it is responsible for complying with any applicable laws and regulations, including, but not limited to laws and regulations related to call recording, data privacy, intellectual property, trade secret, fraud, and music performance rights, in the country or territory where the Avaya product is used.

## **Preventing Toll Fraud**

“Toll Fraud” is the unauthorized use of your telecommunications system by an unauthorized party (for example, a person who is not a corporate employee, agent, subcontractor, or is not working on your company's behalf). Be aware that there can be a risk of Toll Fraud associated with your system and that, if Toll Fraud occurs, it can result in substantial additional charges for your telecommunications services.

## **Avaya Toll Fraud intervention**

If You suspect that You are being victimized by Toll Fraud and You need technical assistance or support, call Technical Service Centre Toll Fraud Intervention Hotline at +1-800-643-2353 for the United States and Canada. For additional support telephone numbers, see the Avaya Support website: <https://support.avaya.com> or such successor site as designated by Avaya.

## **Security Vulnerabilities**

Information about Avaya's security support policies can be found in the Security Policies and Support section of <https://support.avaya.com/security>. Suspected Avaya product security vulnerabilities are handled per the Avaya Product Security Support Flow (<https://support.avaya.com/css/P8/documents/100161515>).

## **Downloading Documentation**

For the most current versions of Documentation, see the Avaya Support website: <https://support.avaya.com>, or such successor site as designated by Avaya.

## **Contact Avaya Support**

See the Avaya Support website: <https://support.avaya.com> for product or Cloud Service notices and articles, or to report a problem with your Avaya product or Cloud Service. For a list of support telephone numbers and contact addresses, go to the Avaya Support website: <https://support.avaya.com> (or such successor site as designated by Avaya), scroll to the bottom of the page, and select Contact Avaya Support

## Contents

Overview .....	8
Context .....	8
Use cases .....	9
Introduction.....	9
Service User Configuration .....	10
Licensing .....	11
Transport Options .....	11
Protocol Description .....	12
Message .....	12
Message .....	12
Subscriptions .....	13
Subscribe .....	13
RequestResponse .....	15
Notify.....	15
NotifyAck .....	16
End the Subscription.....	17
Subscription services .....	19
Subscribe for Presence .....	19
Subscribe for Lines.....	24
Notify Response .....	25
Subscribe User.....	28
Subscribe Queue .....	34
Call Control Notifications.....	39
Modifying the Subscription .....	43
CallControl explicitly for Queue calls .....	48
Actions on Queued calls .....	49
BlindTransfer to a Callflow .....	50
BlindTransfer to a MeetMe Conference.....	51
BlindTransfer to a conference requiring PIN access.....	52
BlindTransfer to a functional queue.....	53
Error codes .....	54
Features Available .....	56
DropCall .....	56

AnswerCall .....	57
HoldCall .....	58
UnHoldCall .....	59
BlindTransfer .....	59
Redirect .....	60
Dial.....	61
Park.....	62
SetupTransfer .....	62
SetupConf .....	63
CompleteTransfer .....	64
CompleteConference.....	65
AddToConference .....	66
MemberFunction.....	67
SetTag .....	68
SetAccountCode .....	68
SetNotes .....	69
PushToEC500 .....	69
GenerateDigits.....	70
ShortCodeAction.....	70
AnswerPage.....	70
ForceClear.....	71
SetAuthCode.....	72
CallRecordingOn/Off.....	72
PrivacyOn/Off .....	73
MuteOn/Off.....	74
SetPriority.....	74
Finish .....	75
Alternative connection methods .....	76
Noframing.....	76
Limits .....	76
Version Compatibility .....	76
Resilient solutions.....	76
What this means .....	76
Resilient app .....	76
Development tools .....	76
SysMonitor .....	77

Getting started with the proto file.....	78
C++ and visual studio.....	78
Java.....	81
Javascript.....	81
Establishing a Websocket connection .....	81
First payload .....	82
Early releases of IP Office .....	82
Change History .....	83

## Overview

This document details a protocol that is supported on IP Office since Release 11.1.0.0. It is designed for ContactCenter-type applications (one application per solution, rather than one application per user), though you can have more than one application connected at a time.

The login credentials are service user credentials rather than telephone user credentials, reflecting that it is a system service.

This interface allows an application to observe call activity on users and queues. Also, to get presence activity on the solution.

For users

You can manipulate some user configuration, perform some call control on users.

For queues

You can manage queue membership, change queue service mode and manipulate calls in queues. Calls in queues can be observed and manipulated even after the call is answered.

This interface is offered by way of Protocol buffers over a websocket. The websocket is rendered directly by IP Office and does not require any additional components (such as oneX portal) to be installed.

## Context

MTCTI3 is an alternative to 3<sup>rd</sup> Party TAPI on IP Office. The 3<sup>rd</sup> party TAPI API for IP Office is limited to Windows only and is not available in secure environments. All functionality that was available on IP Office 3<sup>rd</sup> party TAPI should also be available over MTCTI3, and MTCTI3 includes many additional features.

MTCTI3 is a bit more complex to use, as the Asynchronous nature of the connection to IP Office must be handled by the application. So, for example if "UnPark" is called on TAPI, the interface returns the result (SUCCESS with handle)/FAIL. On MTCTI3, you send an UnPark request and must handle the fact that the result is not available immediately.

In some deployments, it would be possible to have a 3<sup>rd</sup> party TAPI application and a MTCTI3 application both connected to the same PBX. They are not incompatible, and they would work cooperatively.

The MTCTI3 interface allows the application to view IP Office users and Queues, to make live changes to the configuration of Users and Queues related to telephony functionality, and to view and manipulate telephone calls. The single interface can and is expected to handle multiple calls to several users at the same time.

The MTCTI3 interface is feature-rich but can be used for very simple functions like monitoring the Do-not-disturb setting for a user or noting the calling number for incoming calls.

A MTCTI3 application would typically maintain lists of lists. A list of users (and/or queues) and a list of live calls for each user or queue. The lists must be kept updated by Notify messages received asynchronously from IP Office.

Each Notify of a call change is a complete snapshot of the call, and the application needs to compare the new data with the old data to see what, if anything, has changed. This snapshot concept is important. The application does not necessarily see all the transitions of a call, only the current state of a call. So an incoming call which is auto-answered or answered quickly may never be seen in the RINGING state. Applications should not rely on seeing all transient states.

Most calls on IP Office go to/from users or to queues. However, there are some calls that do not – they may go directly to an Auto-attendant or be routed directly out of another trunk. These calls are not visible to the MTCTI3 application.

MTCTI3 is authenticated by secure methods and is trusted with powerful controls. There are no configuration items on IP Office to constrain the MTCTI3 from accessing users or queues, so the writer of the application needs to provide the constraint where appropriate.

Even though this API document starts with a description of the Presence subscription, it is expected that most applications will be using Lines, User and Queue subscriptions, and may not use Presence at all.

## Use cases

A simple use case for MTCTI3 without even subscribing to Call Control is to monitor a set of users on IP Office to see whether they are in Do Not Disturb, or Available to receive calls.

A more complex use case would be to observe a group of users to see how busy they are, and which calls they are making and receiving.

A highly functional use case would be for the MTCTI3 application to take control of a Queue and distribute all calls arriving at that queue to appropriate Agents.

A Contact center application may be in the business of rendering a business-specific desktop UI to a set of Agents or Supervisors. If this application connected to IP Office over MTCTI3, it could add telephony controls to the Agents desktop, to set Available/Not available, Answer calls (by controlling their desk phone or soft-phone), Make calls.

## Introduction

Third party developers would be expected to develop applications using this document as a reference.

The MTCTI3 interface to IP Office is a protocol definition only. The 1<sup>st</sup> competence required of the engineer is to take the protocol definition and make an interface. The work required depends on the platform, language and environment of the application that wishes to use MTCTI3. For some languages, and environments (example JavaScript on Angular), the framework has methods to consume the protocol definition file directly into managed objects. For other languages, the engineer will want to source a 'proto' file compiler, and protobuf encoding and decoding source code. This is what you would expect to have to do as the MTCTI3 application is expected to be a

server application, not a desktop application. Information on how to do this is provided in later chapter “Getting started with the proto file”.

Additionally, once you have the encoding and decoding of messages, you need to be able to connect to IP Office over a secure websocket. Each development environment has a different implementation of websockets, and the engineer must be able to create a websocket to the IP Office web service, authenticate, and then send and receive messages over the websocket.

MTCTI3 requires authentication in the websocket handshake, which is not necessarily available on all HTML5 browser websocket implementations.

The developer needs to have a basic understanding of Users and Queues on IP Office as these are what are being manipulated.

The protocol will work in stand-alone or in SCN environments.

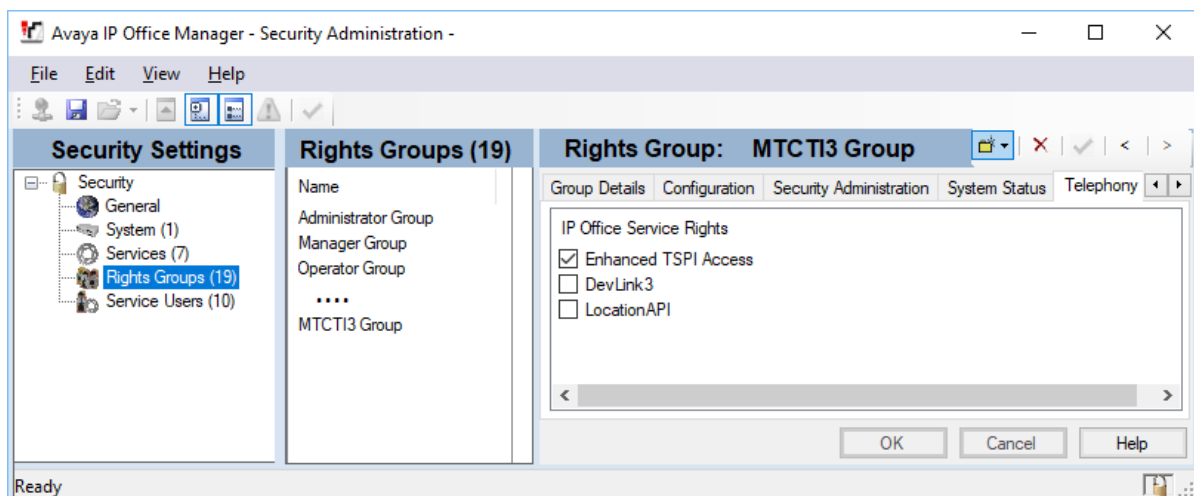
There are several pre-requisites required on the IP Office for the application to successfully connect to an IP Office.

- The IP Office must have sufficient CTI-link pro licences installed for the size of the SCN.
- The IP Office must have “Avaya HTTP Clients only” flag disabled.
- For an SCN solution, the queues that the application is to monitor/control should be configured on the primary PBX, and the application should connect to the primary PBX.
- If connecting to secondary PBX for resilience, the secondary PBX will also require CTI-Link PRO licences. The MTCTI3 on the secondary will not report calls on the primary queues unless the primary queues are failed over, but all user calls and user status are fully accessible on either system.

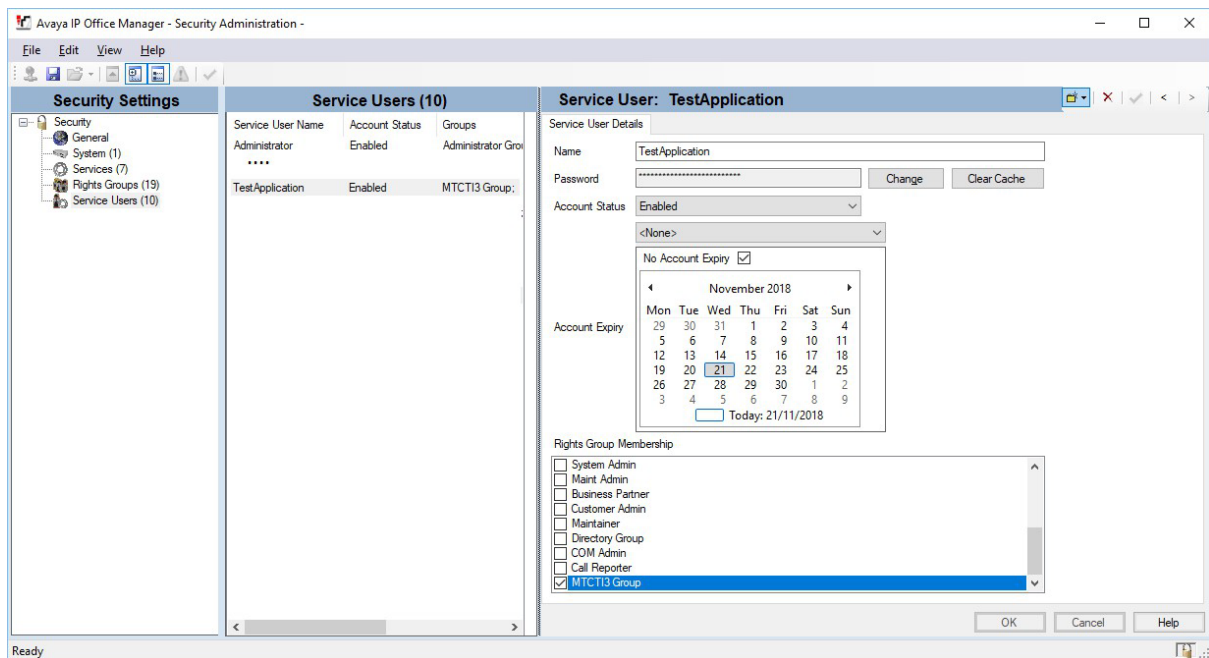
There needs to be configured a Service user + password, who has access rights to “Enhanced TSPI Access” service.

## Service User Configuration

First configure a Service group, with access rights to the “Enhanced TSPI Access” service:



Then create serviceUser account which is a member of this group:



## Licensing

MTCTI3 uses the same licensing as other CTI interfaces on IPOffice:

- The CSTA OpenAPI which can be accessed from oneX portal
- Devlink3 interface
- 3<sup>rd</sup> party TAPI

A fully licensed system allows all these CTI interfaces to work.

Note that there is no partial functionality for a partially licensed system. You need the full set of licences for your deployment, but that allows the application to monitor all users on the SCN from the single connection.

Also note that if you are developing a resilient solution with separate MTCTI3 connections to Primary and Secondary, you need the full set of licences on both Primary and Secondary.

1 x CTILink Pro licence	Standalone IP Office
2 x CTILink Pro licence	Network of up to 5 IP Offices (including this one)
3 x CTILink Pro licence	Network of up to 20 IP Offices (including this one)
4 x CTILink Pro licence	Unlimited Network of IP Offices

## Transport Options

This protocol is available over web-socket only. This protocol is one of several web services that IP Office can deliver. You can only connect to this service over HTTPS. The HTTP variant is disabled on all deployments. This web service is "tpkt/openapi".

The HTTPS service port for IP Office web services defaults to 443 in most environments. However, it is configurable on the Security settings and may be different on cloud deployments.

The credentials required to access the web service will be Service User Name and Service User Password.

These are normally encoded in the Authorization header of the simplified HTTP request below(shown here with the content masked out)

```
GET /tpkt/openapi HTTP/1.1
Connection: Upgrade
Authorization: Basic *****
Upgrade: websocket
Sec-WebSocket-Key:
Sec-WebSocket-Protocol: openapi
Sec-WebSocket-Version: 13
```

Once connected, the payload carried over the Websocket protocol is framed protocol-buffers in either direction

4 octets	0x1 = Framed protocol buffer
N octets	Protocol buffer payload

## Protocol Description

The protocol buffer schema is obtained by compiling the file “ipo\_mtcti3.proto”. This file format is proto3.

For a description of the language, you can search for: “google protocol buffers version 3” on the internet. It is a google sponsored language and messaging format.

The way to use this protocol is for the client (the application) to subscribe to a number of services, and the application will receive notifications on the subscribed services. The client can also send “SubscribeCmd” messages on subscriptions to execute changes.

Each subscription can optionally time out unless refreshed. This is generally the choice of the client application, except in the case of presence subscriptions which must have a specified timeout to refresh.

## Message

### Message

**Message** is the base message of the mtcti-3 protocol and all other service level messages are encapsulated within the **Message**.

```

message Message
{
  oneof payload
  {
    RequestResponse response = 1;
    GeneralCmd generalcmd = 2;
    Subscribe subscribe = 3;
    SubscribeCmd subscribecmd = 4;
    SubscribeEnd subscribeend = 5;
    SubscribeTerminated subscribeterminated = 6;
    GeneralData generaldata = 7;
    Notify notify = 8;
    NotifyAck notifyack = 9;
  }
}

```

One of the payloads should be set in the **Message**.

Fields	Description	Direction
<b>response</b>	Set the payload for response to the Request	IP Office → App
<b>subscribe</b>	Set the payload for Subscribe Request	App → IP Office
<b>subscribecmd</b>	Set the payload for SubscribeCmd Request	App → IP Office
<b>subscribeend</b>	Set the payload for SubscribeEnd request	App → IP Office
<b>subscribeterminated</b>	Set the payload for SubscribeTerminated event	IP Office → App
<b>notify</b>	Set the payload for Notify message	IP Office → App
<b>notifyack</b>	Set the payload for NotifyAck message	App → IP Office
<b>generalcmd</b>	App issues a context-less command	App → IP Office
<b>generaldata</b>	Response to the generalcmd	IP Office → App

## Subscriptions

This section covers how the client should subscribe for the different services, send updates, receive notifications in the subscription. Also ending subscription from the client and Server.

This section covers Subscribe, RequestResponse, Notify and NotifyAck messages.

### Subscribe

This message enables the client to subscribe for one of the subscriptions.

```

message Subscribe
{
  int32 requestid = 1;
  int32 subscribe_id = 2;
  int32 timeout = 3;
  string label = 4;
  oneof payload
  {

```

```

SubscribePresence presence = 10;
SubscribeLines lines = 40;
SubscribeUser user = 41;
SubscribeQueue queue = 42;
SubscribeParkServer parkserver = 43;
SubscribeRefreshWrapper refreshwrapper = 45;
}
}

```

Fields	Description
<b>requestid</b>	ID for the particular Request
<b>subscribe_id</b>	ID for the particular Subscription
<b>timeout</b>	Expiry value for the particular subscription
<b>label</b>	Label for the particular subscription
<b>One of the payloads</b>	
<b>presence</b>	Set the payload for the Presence subscription
<b>lines</b>	Set the payload for the lines subscription
<b>user</b>	Set the payload for an individual user subscription
<b>queue</b>	Set the payload for an individual queue subscription
<b>parkserver</b>	Set the payload for the parkserver subscription
<b>refreshwrapper</b>	Set the payload for the refreshwrapper subscription

#### *Subscribe\_id*

Subscription message should contain the “subscribe\_id” and one of the subscription payloads. “subscribe\_id” will be used in all messages in either direction related to the subscription. For this reason, the “subscribe\_id” should be chosen by the client to be a unique number in the context of the connection.

#### *requestid*

Messages from the client may contain a “requestid”. If a “requestid” is populated, IP Office will send a RequestResponse indicating that the message has been received.

#### *label*

This is an optional string. It is not used by IP Office.

#### *timeout*

“Timeout” value set to zero, or not specified means no expiry. Presence subscriptions should have explicit “Timeout” value and value should set in seconds between 60 and 86400. Units are seconds.

In order to refresh a Subscribe, the client should send a new Subscribe message with the same subscribe\_id, and only containing a new timeout value:

```

Message
{
  subscribe
  {
    subscribe_id=98765
    timeout=3600
    presence

```

```
{
  .....
}
}
```

After (eg) 50 minutes, send a refresh....

```
Message
{
  subscribe
  {
    subscribe_id=98765
    timeout=3600
  }
}
```

This will now run another 3600 seconds before terminating.

## RequestResponse

“RequestResponse” message used to acknowledge the Request with the results.

```
message RequestResponse
{
  int32 requestid = 1;
  int32 result = 2;
  string additional = 3;
}
```

This message is used to acknowledge both Subscribe and General Commands requests.

Fields	Description
<b>requestid</b>	ID of the received Request
<b>result</b>	Success or error code (error codes in Appendix)
<b>additional</b>	Additional details, for example, error reason string. Not currently used.

## Notify

“**Notify**” message uses to send notification to a subscriber to inform on the latest change on the resources on which the Subscriber is interested.

```
message Notify
{
  int32 subscribe_id = 1;
  int32 notify_id = 2;
  string label = 3;
  oneof payload
  {
    NotifyPresence presence = 10;
```

```

NotifyCallControl callcontrol = 14;
NotifyLines lines = 40;
NotifyUser user = 41;
NotifyQueue queue = 42;
NotifyRefreshWrapper refreshwrapper = 45;
}
}

```

Fields	Description
<b>Subscribe_id</b>	ID present in the Subscribe Request
<b>Notify_id</b>	Notify ID added by the IP Office
<b>label</b>	Usually label from Subscribe Request (not currently)
<b>One of the payloads</b>	
<b>presence</b>	Set if Notify is for Presence subscription
<b>callcontrol</b>	CallHandling events if subscribe is one of User, Queue, ParkHandler
<b>lines</b>	Set if Notify is for lines subscription (add/remove users or queues)
<b>user</b>	Set if Notify for User subscription (user status or config)
<b>queue</b>	Set if Notify for Queue subscription (queue status or config)
<b>refreshwrapper</b>	Set if Notify for refreshwrapper subscription

Client should acknowledge Notify message by sending **NotifyAck**, if “notify\_id” present in the received NOTIFY message. Client should ignore NOTIFY message, if one of the payloads is not set, subscription Id does not exist or payload is not expected with “subscribe\_id” mentioned in the Notify message.

### NotifyAck

Client should acknowledge the NOTIFY message by sending NotifyAck message, if “notify\_id” present in the received NOTIFY message. If “notify\_id” is included by IP Office in the Notify message, IP Office does not send next Notify until the last Notify is acknowledged.

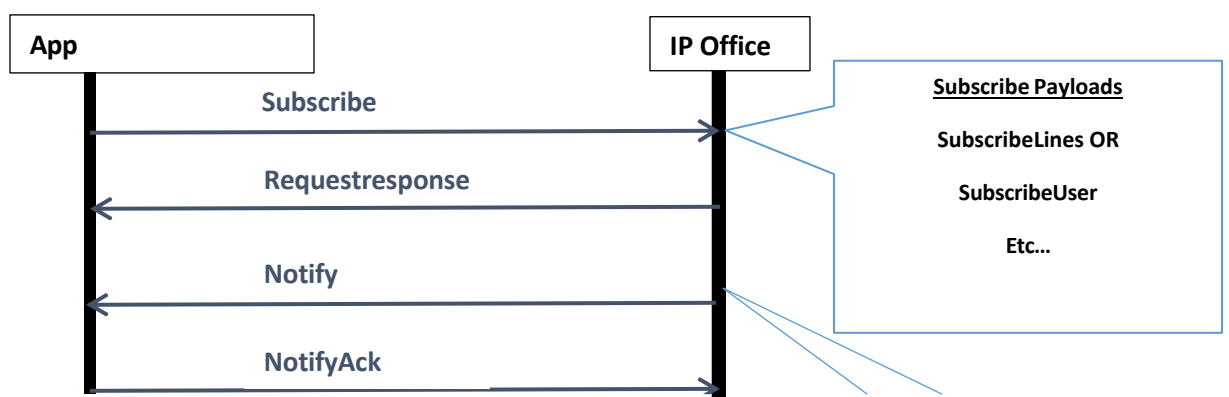
```

message NotifyAck
{
    int32 subscribe_id = 1;
    int32 notify_id = 2;
}

```

Fields	Description
<b>subscribe_id</b>	Corresponding Subscription ID
<b>notify_id</b>	Notify Id from the NOTIFY message

### Subscription message flow



**Figure 1 - Subscribe Message flow**

### End the Subscription

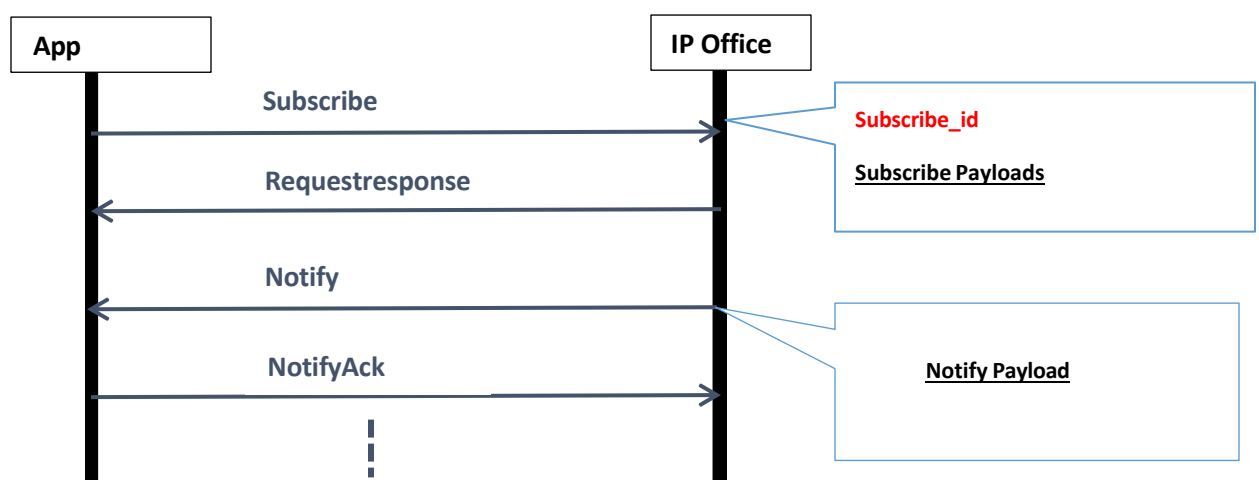
A subscription can be terminated at any time by the App sending a Subscribe-End request. Subscribe-End request should contain “subscribe\_id” of the subscription that needs to be terminated.

### SubscribeEnd

```
message SubscribeEnd
{
  int32 requestid = 1;
  int32 subscribe_id = 1;
  string reason = 3;
}
```

Fields	Description
<b>request_id</b>	Request ID
<b>subscribe_id</b>	Subscription ID (required)
<b>reason</b>	Reason string. Not functional, but may be added to a report in IP Office.

### Subscribe End message flow



## Terminate the Subscription

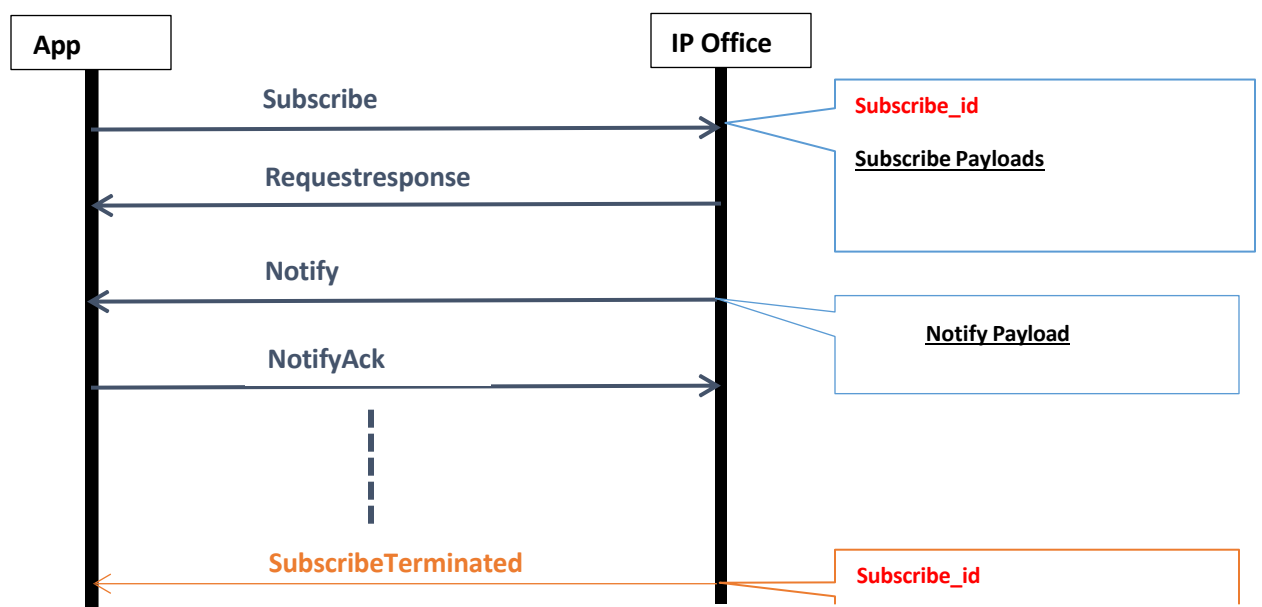
IP Office can terminate a subscription asynchronously by sending SubscribeTerminated message. “subscribe\_id” indicates the subscription that is terminated by IP Office.

### *SubscribeTerminated*

```
message SubscribeTerminated
{
    int32 subscribe_id = 1;
    string reason = 2;
}
```

Fields	Description
<b>subscribe_id</b>	Subscription ID
<b>reason</b>	Reason string (not currently populated)

### *Subscription terminated message flow*



## Subscription services

Each subscription service is independent and can be unsubscribed individually. You should only subscribe to the services that you need for your application.

The following services are available:

Service	Description	Notes
Presence	Each presence subscription contains a list of presentities to watch.	You can have several subscriptions at a time, each containing a different list of presentities.
Lines	This maintains an updated list of all the users on the SCN and queues on the PBX	Only one of these is allowed
User	This is a subscription for a particular user. It renders individual user status, and optionally the telephony calls presenting on the user.	Choose users out of the lines list.
Queue	This is a subscription for an individual Queue. It renders Queue configuration, and optionally calls that are being handled on the queue.	Choose queues out of the lines list, or if you know the queue name, you can just subscribe by queue name.
ParkServer	This subscribes to the activity on the ParkServer. That is showing calls which are currently parked..	Only one of these is allowed
RefreshWrapper	This is a little utility to help group notifications on different subscriptions into atomic events. It is not watertight, but it can be useful.	Only one.

### Subscribe for Presence

Client should set **SubscribePresence** payload in the **Subscribe** request message for the presence subscription, with a table of presentities to monitor. On successful subscription, IP Office sends **RequestResponse** for the request.

SubscribePresence may not be of major interest. However, it was the first service to be implemented.

### SubscribePresence

This message is a payload of **Subscribe** message.

```
message SubscribePresence
{
    repeated Presentity entry = 1;
    int32 flags = 2;
}
```

Fields	Description
<b>entry</b>	List of the Presentities
<b>flags</b>	Flags is a bitfield of options 0x01: include unread voicemail counter in presentity 0x02: Do not change app presence to 'Offline' when all apps are disconnected

### Presentity

This message is an element of **SubscribePresence** and carries details of presentity. Presentity can be a phone number, a SIP address, an Email address or a UniqueID. Each presentity should be assigned a **local reference ID (LREF)** to reduce the size of the NOTIFY updates (though this is not mandatory).

Note that presence subscription is really an alternative to subscribing to all the users individually. If you subscribe to all the user lines, you get the basic part of the presence information from each line.

```
message Presentity
{
    string presentity = 1;
    int32 lref = 2;
}
```

Fields	Description
<b>presentity</b>	Presentity can be phone number, SIP URI or Email address.
<b>lref</b>	lref is a local reference ID of the presentity

### NotifyPresence

**NotifyPresence** is a payload of **Notify** message. From the provided presentity list, set of all IP Office users will be searched. IP Office will NOTIFY all presentities in the monitor list in **NotifyPresence** payload. All new presentities will have their initial state NOTIFY as soon as they are added to the Subscribe. Afterwards, only changed presentities will be NOTIFIED. This will be indicated in "full" field.

```

message NotifyPresence
{
    int32 full = 1;
    repeated NotifyPresentity entry = 2;
}

```

Fields	Description
<b>full</b>	Indicates whether presentities list is full or just changes.
<b>entry</b>	Holds the list of Presentity elements

### *NotifyPresentity*

Each Notify will contain all the tracked fields for that presentity. Tracked fields are carried in **NotifyPresentity** message, which is an element of **NotifyPresence**.

```

message NotifyPresentity
{
    string presentity = 1;
    int32 lref = 2;
    int32 sac = 3;
    int32 phonestate = 4;
    Absence absence = 5;
    string app = 6;
    bool fwdu = 7;
    int32 vmunread_messages = 8;
    bool noapphandler = 9;
}

```

Fields	Description
<b>presentity</b>	Presentity identifier, provided by Client
<b>lref</b>	lref of Presentity , provided by Client. If this is specified, then the presentity identifier above will not need to be included.
<b>sac</b>	DND telephony status of the presentity
<b>phonestate</b>	Phone state of the presentity
<b>absence</b>	Absence text set by the presentity
<b>app</b>	App presence set by the presentity
<b>fwdu</b>	Presentity has forwarding enabled
<b>vmunread_messages</b>	Presentity has unread voicemail messages (dependent on subscribe flags = 0x01)
<b>noapphandler</b>	There are no applications that can control app presence. (dependent on subscribe flags = 0x02). If this subscribe flag is *not* set, then when there are no applications, the “app” string will show “Offline”

- Absence Text maximum length is set to 21.

- Application presence maximum length is set to 34.
- “sac” is set to non-zero, if DND is enabled for the presentity.

Phone state ID	State of the phone
0	Idle
1	Dialling
2	Ringing
3	ACW
4	Connected
5	Logout
6	Fault
7	Recover
8	Unknown state

**Table 1 - Phone states**

### *UpdatePresence*

You can add and remove individual presentities from a subscription using an Update message. An example is below.

1<sup>st</sup> subscribe...

```

Message
{
  subscribe
  {
    subscribe_id=98765
    timeout=3600
    presence
    {
      entry
      {
        presentity=201
        lref=5678
      }
      entry
      {
        presentity=202
        lref=5679
      }
      entry
      {
        presentity=203
        lref=5680
      }
    }
  }
}

```

Later update...

```
Message
{
  subscribecmd
  {
    subscribe_id=98765
    presence
    {
      add
      {
        presentity=204
        lref=5681
      }
      add
      {
        presentity=205
        lref=5682
      }
      remove
      {
        lref=5678
      }
    }
  }
}
```

You will get an immediate notification of the states of the new presentities.

### Absence

Absence will only be present if there is an Absence message set. When set, the message has the following elements:

```
message Absence
{
  int32 msg = 1;
  string str = 2;
}
```

Fields	Description
msg	Message type.
str	Holds the Absence text.

If the msg is set to 11, the whole note will be contained in “str”.

If the Absence text is set on an IP Office desk handset, the following values may be set in msg:

Fields	Description
--------	-------------

1	“ON HOLIDAY UNTIL” + str
2	“WILL BE BACK” + str
3	“AT LUNCH UNTIL” + str
4	“MEETING UNTIL” + str
5	“PLEASE CALL” + str
6	“DON’T DISTURB UNTIL” + str
7	“WITH VISITOR UNTIL” + str
8	“WITH CUSTOMER UNTIL” + str
9	“BACK SOON”
10	“BACK TOMORROW”
11	str

Note that msg=11 and str ="" (or null) is a blank string. This is not the same as Absence not set and should be avoided.

### Subscribe for Lines

The lines subscription gives a list of Users and Queues, and sufficient information to subscribe to the individual user or queue. You will automatically get updates when a User or Queue is added or removed from the network.

```
message LinesUser
```

```
{
  bytes guid = 1;
  string extn = 2;
  string name = 3;
}
```

```
message LinesQueue
```

```
{
  bytes guid = 1;
  string extn = 2;
  string name = 3;
}
```

```
message NotifyLines
```

```
{
  int32 flags = 1;
  repeated LinesUser adduser = 2;
  repeated LinesQueue addqueue = 3;
  repeated LinesUser deleteuser = 4;
  repeated LinesQueue deletequeue = 5;
  repeated LinesUser changeuser = 6;
  repeated LinesQueue changequeue = 7;
}
```

```
message SubscribeLines
```

```
{
  int32 flags = 1;
```

```
}
```

You can choose whether to just get notification for Users, or just get notification for Queues via the flags in the SubscribeLines. field with is a bit array. You can also see availability for the ParkHandler, though this is normally always available.

Here is an example sequence:

```
Message
{
  subscribe
  {
    requestId=1
    subscribe_id=5555
    timeout=3600
    lines
    {
      flags=7
    }
  }
}
```

You get a RequestResponse, because the Subscribe had a requestId

```
Message
{
  requestresponse
  {
    requestId=1
    result=0
  }
}
```

You should have only one Subscription to “Lines”. The IP Office will notify you of all the lines (Users and Queues), which you subscribe to separately.

Meaning of “Flags”

Bit 0	List all users
Bit 1	List all groups
Bit 2	Report park server availability

### Notify Response

The first Notify will contain a list of all the Users and Queues in the system.

```
Message
```

```
{
  notify
  {
    Subscribe_id=5555
    Notify_id=1
    lines
    {
      flags=3
      adduser
      {
        guid=DC51BA0008A311DD80540050569F6EF8
        extn=2002
        name=Bergcamp
      }
      adduser
      {
        guid=DC51BA0008A311DD80550050569F6EF8
        extn=2003
        name=Viera
      }
      adduser
      {
        guid=DC51BA0008A311DD80560050569F6EF8
        extn=2004
        name=Lampard
      }
      adduser
      {
        guid=DC51BA0008A311DD80570050569F6EF8
        extn=2005
        name=Scholes
      }
      adduser
      {
        guid=DC51BA0008A311DD80580050569F6EF8
        extn=2006
        name=Foster
      }
      addqueue
      {
        guid=21218C000AF711DD80BC0050569F6EF8
        extn=2501
        name=IP Office Sales
      }
      addqueue
      {
        guid=372E6C800AF711DD80BD0050569F6EF8
        extn=2502
        name=Service Queue
      }
      addqueue
```

```

    {
      guid=8461A88028D811DD807C0050569F5DC5
      extn=8009
      name=Supervisors
    }
    addqueue
    {
      guid=5E883D8028D911DD817B0050569F5DC5
      extn=8008
      name=Agents
    }
  }
}
}
}

```

Because there was a notify\_id in the Notify, the application should send a NotifyAck

```

Message
{
  notifyack
  {
    subscribe_id=5555
    notify_id=1
  }
}

```

Meaning of Notify “Flags”

Bit 0	Park server can be subscribed
Bit 1	Meaning not described here
Bit 2	Meaning not described here.

Then subsequently when an administrator Adds or removes a set of users, there will be another Notify

```

Message
{
  notify
  {
    subscribe_id=5555
    notify_id=2
    lines
    {
      flags=3
      adduser
    }
  }
}

```

```

        guid=B8E477802F2511DE805A0050569F5DC5
        extn=3500
        name=OJordan
    }
    adduser
    {
        guid=09ED8B803BA111DE805C0050569F5DC5
        extn=3501
        name=NLow
    }
    adduser
    {
        guid=30125B803BA111DE811F0050569F5DC5
        extn=3502
        name=AVilani
    }
    adduser
    {
        guid=56382B803BA111DE81EB0050569F5DC5
        extn=3503
        name=KRihanoff
    }
}
}

```

If the administrator changes the name or extn number of a user, you will get a changeuser with the same GUID ....

```

Message
{
    notify
    {
        subscribe_id=5555
        notify_id=2
        lines
        {
            flags=3
            changeuser
            {
                guid=B8E477802F2511DE805A0050569F5DC5
                extn=3550
                name=OJordan
            }
        }
    }
}

```

### Subscribe User

Using the lines subscription, the allocation now has a table of users and queues. From this, you could just subscribe to Presence, but if you want to perform functions on a user, you need to Subscribe to

each user individually. Typically, the application may start a large number of subscriptions at this point.

```
message SubscribeUser
{
    bytes guid = 1;
    int32 flags = 2;
    int32 ccflags = 3;
}
```

You must use the GUID out of the lines table in the Subscribe:

```
Message
{
    subscribe
    {
        requestid=2
        subscribe_id=7000
        user
        {
            guid=DC51BA0008A311DD80530050569F6EF8
            flags=0x1
            ccflags=0x19
        }
    }
}
```

The subscribe\_id must be a new value for each user subscription.

There are effectively two parallel subscriptions being enabled here. (User config) + (User call control). If you do not need to subscribe to call control, then do not set the ccflags. Then you will just get the basic user configuration.

Meaning of “flags” in user

Bit 0 (0x01) MAILBOX	Include voicemail box message counters
Bit 1 (0x02) FWD	Include Forwarding details
Bit 2 (0x04) APP	Include Application presence
Bit 3 (0x08) ABSENCE	Include Absence (when enabled)
Bit 11(0x800) ACTIVAPP	If this is set, then this subscription actively counts as an application that can edit the app field. (So Equinox phones will not say ‘Offline’ on this presentity while this subscription is registered) This flag can be actively changed using “makelive” or “makedead” booleans on the UpdateUserApp payload of an UpdateUser message.
Bits 4..10 and 12..31	Not described here

Meaning of “ccflags” in user. These flags define the level of detail you will get back in callinfo events.

A good setting for regular applications would be 0x99

Bit 0 (0x01)	Primary callcontrol data
--------------	--------------------------

Bit 1 (0x02)	Local and remote devices
Bit 2 (0x04)	Additional simultaneous targets
Bit 3 (0x08)	Conference membership
Bit 4 (0x10)	Dial info
Bit 5 (0x20)	Extended Trunk detail
Bit 6 (0x40)	Extended Queue information
Bit 7 (0x80)	Language, privacy

From the User config subscription, you get back a “user” notify.

```

Message
{
  notify
  {
    subscribe_id=7000
    notify_id=1
    user
    {
      extn=2002
      name=Bergcamp
      email=dbergcamp@denmarklegends.com
      language=dan
      voicemail=1
      mailbox
      {
      }
    }
  }
}

```

#### Fields in User Notify

extn	User number in the dial plan
name	User name
fullname	User full name
katakananame	User katakana name (Japan only)
email	User email address
featuresavailable	Actions that are allowed
language	Locale code
dnd	Do not disturb
barred	User is barred from making external calls
xdirectory	User is ex-directory
voicemail	User has voicemail enabled
loggedinextn	User is logged in to this extension
extnfault	User does not have a working phone
loggedoff	User is not logged in anywhere
absence	Absence info
mailbox	Summary content of the voicemail mailbox
app	Application presence string
noapphandler	There is no application registered that can edit the app field
ec500	Mobile twinning enabled
fwdu	Forward Unconditional
fwdb	Forward on busy
fwdna	Forward on no answer

fwdhg	Hunt group calls follow fwdu rule
fwdunumber	Forward Unconditional destination
fwdbnumber	Forward busy destination, if different from fwdunumber.
fwdtovm	Forward Unconditional destination is voicemail
offswitch	User is allowed to set forward number to off-switch

Featuresavailable

This is a bit-field

Bit 0 (0x00000001) VALID	Means this field is populated (not present in older versions)
Bit 1 (0x00000002) SAC	Can Set or Unset Do Not Disturb
Bit 2 (0x00000004) ABSENCE	Can change absence text
Bit 3 (0x00000008) EC500	Can change mobile twinning setting
Bit 4 (0x00000010) LOGIN	Can hot-desk to another extension
Bit 5 (0x00000020) SHORTCODE	Allowed to send shortcodes
Bit 6 (0x00000040) MAKECALL	Allowed to MakeCall
Bit 7 (0x00000080) FWDU	Allowed to set Forward Unconditional
Bit 8 (0x00000100) DFOI	(reserved)
Bit 9 (0x00000200) APP	Allowed to change the application presence string

If one of these fields changes, you will get a new notify.user on this same subscription.

If any bits are set in ccflags, you also get a Callcontrol subscription on this same subscribe\_id. For example, an incoming call looks like:

```

Message
{
  notify
  {
    subscribe_id=7000
    callcontrol
    {
      refreshinstance=1
      callinfo
      {
        callid=1
        state=RINGING
        direction=INBOUND
        gcid = 01 C0 A8 2A 0B 00 00 03 EC
        featuresavailable=0x23 DROP ANSWER REDIRECT
        callingparty
        {
          number=61001
          name=Agent 61001
          nametype=5
        }
      }
    }
  }
}

```

Call Control commands and notifications are described later.

### Update User

You can issue commands on the user subscription to change the user's configuration

```
message UpdateUser
{
    UpdateUserSAC sac = 1;
    UpdateUserAbsence absence = 2;
    UpdateUserEC500 ec500 = 3;
    repeated ShortCodeData shortcodedata = 4;
    UpdateUserLoginExtn loginextn = 7;
    UpdateUserApp app = 8;
    UpdateUserFwdU fwdU = 9;
}
```

The commands available are not very complex, but you can also send ShortCodeData which enables more control.

sac	Enable/disable DND
absence	Set/clear Absence text
ec500	Set/Unset Mobile twinning.
shortcodedata	Other configuration options
loginextn	Hot-desk user to a different extension
app	Change user's application presence
fwdU	Change users forward unconditional settings including On/Off, destination

### Set DND

```
Message
{
    subscribecmd
    {
        subscribe_id=7000
        user
        {
            sac
            {
                set=1
            }
        }
    }
}
```

### Set Voicemail-box content

This is a bit of a random thing to include here but is a useful trick for certain applications that want to offer voicemail services instead of VMPro..

If IP Office is providing the voicemail through (eg) VMPro, then the voicemail server will update the user with message counts: newmessages, oldmessages, savedmessages.

However, if voicemail is being provided by a client application, it is possible for the client application to show the mailbox content through this interface. Note that the IP Office view of the mailbox content may occasionally clear the values, and the application should then re-assert the correct numbers.

This interface is superior to using “DisplayMsg”, as it should show the detail on all interfaces, not just on the phone display.

You use shortcode 70 to set voicemail content, and the shortcodeval string must be in the format ‘;’ (semicolon) followed by \$newmessages,\$oldmessages,\$savedmessages

```
Message
{
  subscribecmd
  {
    requestid=3
    subscribe_id=7000
    user
    {
      shortcode
      {
        shortcode=70
        shortcodeval=;3,0,0
      }
    }
  }
}
```

Results in a notify (provided you set bit 0 in the subscribe):

```
Message
{
  notify
  {
    subscribe_id=7000
    notify_id=2
    user
    {
      extn=61000
      name=User 61000
      language=eng
      voicemail=1
      mailbox
      {
        newmessages=3
      }
    }
  }
}
```

```
}  
}
```

#### Other shortcodes

Shortcode integer	Parameter string	Action
0		Set forward unconditional
1		Unset forward unconditional
2		Set forward on busy
3		Unset forward on busy
4		Set forward no answer
5		Unset forward no answer
6	destination	Set forward number
7		Set DND
8		Unset DND
9	exception	Set DND exception
10		Clear all DND exceptions
17		Voicemail on
18		Voicemail off
94	Time in seconds	Set noAnswer timeout
109	\$set,\$val,\$str	SetAbsentText
114	destination	Set FwdBusy number
166		Set EC500
167		Unset EC500

#### Subscribe Queue

Using the lines subscription, the allocation now has a table of users and queues. From this, you could just subscribe to Presence, but if you want to perform functions on a queue, you need to Subscribe to each queue individually.

```
message SubscribeQueue  
{  
    bytes guid = 1;  
    int32 flags = 2;  
    int32 ccflags = 3;  
    string name = 4;  
}
```

You should use the GUID out of the lines table in the Subscribe:

```
Message  
{  
    subscribe  
    {  
        requestid=2
```

```

    subscribe_id=7500
    queue
    {
        guid=DC51BA0008A311DD80530050569F6EF8
        flags=0x3
        ccflags=0x40039
    }
}
}

```

In some applications, where all you want to do is control a single queue whose name you know, you do not need to subscribe to the lines table to extract the guid. You just subscribe to the queue by name:

```

Message
{
    subscribe
    {
        requestid=2
        subscribe_id=7500
        queue
        {
            flags=0x3
            ccflags=0x40039
            name=Sales
        }
    }
}
}

```

Meaning of “flags” in queue

Bit 0	Include voicemail box message counters
Bit 1	Include huntgroup members list
Bits 2..31	Not described here

Queue Notify

```

message NotifyQueue
{
    enum ServiceMode
    {
        Unset = 0;
        Active = 1;
        NS = 2;
        OOS = 3;
    }
    string extn = 2;
    string name = 3;
    string katakananame = 4;
    string email = 5;
    int32 ringmode = 6;
}

```

```

int32 noanswertime = 7;
bool voicemail = 8;
int32 voicemailtime = 9;
ServiceMode servicemode = 10;
QueueMembers queuemembers = 11;
Mailbox mailbox = 12;
}

```

extn	Number in the dial plan
name	Queue name
email	Queue email
ringmode	0 = ringidle 1 = sequential 2 = group
noanswertime	Time in seconds to ring around agents
voicemail	Voicemail enabled
voicemailtime	Time in seconds before call goes to voicemail
servicemode	Active, Night service or OOS
queuemembers	Agents in this queue
mailbox	Contents of queue's voicemail box.

Mailbox is included if flags bit 0 is set in the subscribe.queue.

```

message Mailbox
{
    int32 newmessages = 1;
    int32 oldmessages = 2;
    int32 savedmessages = 3;
}

```

QueueMembers is included if flags bit 1 is set in the subscribe.queue

```

message QueueMember
{
    string extn = 1;
    bool disabled = 2;
}

message QueueMembers
{
    repeated QueueMember member = 1;
}

```

This is a list of all agents in the queue, and whether they are currently disabled (not accepting calls)

There are a set of commands for administering the Queue configuration.

```

message UpdateQueue
{
    SetServiceMode setservicemode = 1;
    SetVoicemail setvoicemail = 2;
    QueueMembers members = 3;
    repeated QueueMember addmember = 4;
    repeated QueueMember deletemember = 5;
}

```

You can either specify the entire members list, or you can add or remove members from the list individually.

You can also use addmember to change a member status from enabled to disabled in the queue. “addmember” is an update if the member already exists.

#### *Call Control notifications on a Queue*

This is covered in more detail in the Call Control section, but here is a very brief description.

If you set bit 0 of “ccflags” in the subscribe.queue you will automatically get Notifications for calls coming into a Queue. These calls can be tracked and manipulated.

```

Message
{
    notify
    {
        subscribe_id=7500
        callcontrol
        {
            refreshinstance=1
            callinfo
            {
                callid=1
                state=RINGING
                direction=INBOUND
                featuresavailable=0x6060E0B1 DROP BLINDXFER REDIRECT PARK TAG ACCT FORCECLEAR
AUTH PRIO FINISH
                callingparty
                {
                    number=01707123456
                    name=Avaya Test trunk
                    nametype=9
                }
                queuedetail
                {
                    number=98765
                    name=Agents
                    priority=1
                }
                calldata
                {
                    language=eng

```



## Call Control Notifications

User, Queue and ParkHandler can include implicit Call Control subscriptions depending on the setting of the “ccflags” field in the subscription.

### *NotifyCallControl*

**NotifyCallControl** is a payload of **Notify** message.

```
message NotifyCallControl
{
    repeated CallInfo callinfo = 2 ;
    repeated CallLost calllost = 3;
}
```

Notify contains updates of each call appearance (callinfo) until the call appearance is ended, when a Notify will be generated with calllost.

One Notify can contain multiple callinfo updates, and multiple calllost. A single Notify represents a single event, so after a Transfer complete you would probably get calllost for both the CallOnHold and the Assistant call in the same payload.

The table of callinfo does not necessarily contain all the call appearances, only those with reportable changes.

A NotifyCallControl report represents the current information about a call appearance. You are not guaranteed to see every transition phase of a call that moves quickly between phases. So, for example a call that is made and auto-answered may transition instantly to CONNECTED state. You will not get a Notify for all the intermediate phases.

### *CallInfo*

**CallInfo** is a payload of **NotifyCallControl**

```
message CallInfo
{
    enum State
    {
        UNKNOWN = 0;
        DIALTONE = 1;
        DIALLING = 2;
        DIALLED = 3;
        RINGING = 4;
        RINGBACK = 5;
        CONNECTED = 6;
    }
}
```

```

ONHOLD = 7;
ONHOLDPENDTRANSFER = 8;
ONHOLDPENDCONF = 9;
DISCONNECTED = 10;
BUSY = 11;
FAILED = 12;
WAITINGFORACCT = 13;
WAITINGFORAUTH = 14;
WAITINGFORLINE = 15;
REMINDER=16;
AFTERCALLWORK=17;
RINGINGDIVERT=18;
RINGINVOICEMAIL=19;
ANSWEREDBYVOICEMAIL=20;
LEAVINGVOICEMAILMESSAGE=21;
QUEUEING=22;
RETARGETING=23;
}
enum Direction
{
    UNDEFINED = 0;
    OUTBOUND = 1;
    INBOUND = 2;
    PICKUP = 3;
}
enum FailedCause
{
    UNSET = 0;
    UNSPECIFIED = 1;
    UNALLOCATEDNUMBER = 2;
    REJECTED = 3;
    NUMBEROOO = 4;
    NETWORKOOO = 5;
    BARRED = 6;
    NOCHANNEL = 7;
    NOACCOUNTCODE = 8;
    NOAUTHCODE = 9;
    NOLICENCE = 10;
    LOCALRESOURCES = 11;
    BANDWIDTH = 12;
    COMPATIBILITY = 13;
    CANTRECORD = 14;
    NORESPONSE=15;
}
int32 callid = 2;
int32 referencecallid = 3;
int32 relatedcallid = 4;
State state = 5;
Direction direction = 6;
bool activeheld = 7;
bytes gcid = 8;

```

```

int32 featuresavailable = 9;
string calledparty = 10;
PartyInfo callingparty = 11;
PartyInfo connectedparty = 12;
PartyInfo originalcalledparty = 13;
string tag = 14;
string accountcode = 15;
bool mute = 16;
FailedCause failedcause = 17;
int32 featuresavailable2 = 18;
bool recording = 19;
string parkslot = 20;
Absence absence = 21;
bool recordingpaused = 22;
repeated ConferenceMember conferencemember = 52;
DialInfo dialinfo = 53;
TrunkInfo trunkinfo = 54;
QueueInfo queueinfo = 55;
CallData calldata = 56;
Note notes = 62;
Targets targets = 67
}

```

Fields	Description
<b>Callid</b>	Call identifier, provided by IP Office. It has uniqueness only within this subscription.
<b>Referencecallid</b>	Reference identifier provided by the MTCTI app. If call was made using MakeCall, this is the reference provided in the MakeCall. MTCTI App may assert it or change it at any time using an Update.
<b>Relatedcallid</b>	When a call is an Assistant call (eg during an Assisted Transfer), then this is the callid of the call on-hold pending transfer
<b>State</b>	Q.931 style state of the call
<b>Direction</b>	If the device receives a call and is ringing, then that is INCOMING. If the device makes a call and hears RingBack Tone from the far end, that is OUTGOING. If the call has been established by Call Pickup, Call Steal, UnPark, that is PICKUP.
<b>Activeheld</b>	If the person you were talking to has put you on hold, so you are listening to holdmusic, that is activeheld
<b>Gcid</b>	Global call Identity of this call. When a call is made between two parties, they will both see the same gcid. It is not unique across reboots.
<b>featuresavailable</b>	This is a bitfield of CallFunction Updates that may be effective at this time for this call. See final section.
<b>Calledparty</b>	Usually for outbound calls, this is the number that was called.
<b>Callingparty</b>	For incoming calls, this describes the caller.
<b>Connectedparty</b>	For calls where the other end is defined, this describes the other end
<b>originalcalledparty</b>	For incoming calls which have not arrived directly at this user (diversion or huntgroup), this describes the target of the call
<b>Tag</b>	This is a text label which has been attached to this call
<b>Accountcode</b>	When an account code is attached to a call, and the account code is not hidden, it will be presented here.
<b>Mute</b>	Mute is not always available. In IP Office it is not normally possible to mute a call in the PBX. (It has to be muted on the handset / application itself). It is normally possible to mute a call into a Conference.

	Also if 'mute' is set on the handset, this is not going to reflect in this field. This field will only reflect the 'mute' status if it is a controllable scenario.
<b>Failedcause</b>	When trying to make a call, and the call fails, it will report state = FAILED. The reason why the call failed will be in this field.
<b>featuresavailable2</b>	Additional bitfield of featuresavailable. None defined
<b>Recording</b>	[for user] If this call is being recorded under the control of the user, then this field will be set. This does not reflect system recording for which the user does not have visibility or control. [for queue] This indicates whether the call is being recorded by the system.
<b>parkslot</b>	For ParkHandler only, this identifies the parkslot that the call is occupying.
<b>Absence</b>	For calls on a User, this shows the FAR END Absence text. (So the person you are calling)
<b>recordingpaused</b>	This is the paused state of any system call recording. Even though an agent cannot normally control whether a system call recording is in progress, they may be able to control the recording-paused state
<b>conferencemember</b>	This is a repeating list of all the other parties in a conference.
<b>Dialinfo</b>	When making an outbound call, this contains details of what is being dialled, what type of call is being made, whether the display should be suppressed because it contains authorization codes, whether we are withholding our identity.
<b>queuedetail</b>	If the call is in a queue, this gives the detail about the queue and the priority of the call in the queue.
<b>Calldata</b>	This contains a motley set of ancilliary details about the call, like nominal language of the conversation, whether the call is private etc.
<b>Targets</b>	(subscribe.queue call only) This lists the nominal ringing targets for a call which is in the RINGING, QUEUEING or RETARGETING states. Note that with IP Office an Agent will pick up the longest waiting call on a queue, even if apparently on the list of targets for a different call.

### CallLost

**CallLost** is a payload of **NotifyCallControl**

```
message CallLost
{
    int32 callid = 1;
    int32 referencecallid = 2;
    int32 reason = 3;
    bool thisenddropped = 4;
    string description = 5;
}
```

Fields	Description
<b>callid</b>	Call identifier, provided by IP Office. It has uniqueness only within this subscription.
<b>reference_callid</b>	Reference identifier provided by Equinox. If call was made using MakeCall, this is the reference provided in the MakeCall. Equinox may assert it or change it at any time using an Update. If a MakeCall fails instantly, and for some reason there is not a stable FAILED state, you may never see a CallInfo for the call, only the CallLost. In this case, the application will have to match the reference_callid with the failed call attempt.
<b>reason</b>	Regular reason codes: 16 = Normal

<b>thisenddropped</b>	For a mature call, this tells the application which end terminated the call.
<b>description</b>	Not suitable for presenting to the phone UI, as it will not be a localized string. May contain useful information, or not.

## Modifying the Subscription

From the point of view of CallControl subscription, modifying the subscription means exercising call control – making calls, dropping calls etc.

### UpdateCallControl

**UpdateCallControl** is a payload of **SubscribeCmd** message. This message carries the commands for manipulating calls.

```
message UpdateCallControl
{
    int32 callid = 1;
    int32 referencecallid = 2;
    MakeCall makecall = 3;
    CallFunction callfunction = 4;
    UnParkCall unparkcall = 6;
}
```

Fields	Description
<b>callid</b>	The callid assigned by IP Office to this call
<b>referencecallid</b>	The callid assigned by Application to this call. One of referencecallid or callid must be populated.
<b>makecall</b>	Payload description for making a new call. referencecallid must be populated, and callid must not be populated.
<b>callfunction</b>	Commands to manipulate calls already in existence.
<b>unparkcall</b>	Payload description for unparking a call. referencecallid must be populated, and callid must not be populated. When parking and unparking a call, the callid will not be the same. A different number will be assigned on UnPark.

### MakeCall

**MakeCall** is a payload to initiate an outbound call.

```
message AdvancedMakeCall
{
    string accountcode = 1;
    string authcode = 2;
    string tag = 3;
```

```

bool withholdcli = 5;
bool privacy = 6;
string madn = 7;
bool allowcli = 8;
string explicitcli = 10;
}

message MakeCall
{
    string target = 1;
    int32 type = 2;
    AdvancedMakeCall advanced = 3;
}

```

Fields	Description
<b>target</b>	The dialled string. If empty, then call would normally transition to DIALTONE.
<b>type</b>	There are certain values for this field which should be used carefully: 104 = Page Call 105 = Forcefeed 106 = Intrude 108 = Pickup 109 = CampOn (don't allow call to go to voicemail) 116 = Dial a MeetMe conference A normal call should not have this specified.
<b>advanced</b>	This should only be included if you want to add complex attributes to the call you are making. accountcode = Account code to assign to this call authcode = Authorization code to assign to this call tag = Text label to attach to the call withholdcli = make call anonymously privacy = do not allow others to intrude on this call madn = Specify call origin for campaign call allowcli = Reveal CLI on calls from phones which are normally configured as hide CLI. explicitcli = change the CLI of the outgoing call.

### **CallFunction**

Call manipulation controls. The call is selected by the callid or reference\_callid of the UpdateCallControl parent payload.

```

message CallFunction
{
    enum Action
    {
        None = 0;
        DropCall = 1;
        AnswerCall = 2;
        HoldCall = 3;
    }
}

```

```

UnHoldCall = 4;
BlindTransfer = 5;
Redirect = 6;
Dial = 7;
Park = 8;
SetupTransfer = 9;
SetupConf = 10;
CompleteTransfer = 11;
CompleteConf = 12;
AddToConf = 13;
MemberFunction = 14;
SetTag = 15;
SetAccountCode = 16;
Unused16 = 17;
PushToEC500 = 18;
GenerateDigits = 19;
Unused20 = 20;
Unused21 = 21;
Unused22 = 22;
ForceClear = 23;
SetAuthCode = 24;
CallRecordingOn = 25;
CallRecordingOff = 26;
PrivacyOn = 27;
PrivacyOff = 28;
MuteOn = 29;
MuteOff = 30;
Unused31 = 31;
AgentRecordingControl = 32;
Unused33 = 33;
SetPriority = 34;
Finish = 35;
}
Action action = 1;
string arg1 = 2;
MemberFunctionData memberfunctiondata = 4;
repeated CallInstance callinst = 5;
AgentRecording agentrecording = 6;
}

```

Fields	Description
<b>action</b>	Command to perform on the call appearance
<b>arg1</b>	Text argument that goes with certain commands, like Dial
<b>memberfunctiondata</b>	Rich data to go with MemberFunction (manipulating conference members)
<b>CallInstance</b>	In the case of CompleteTransfer, you would normally transfer the relatedcallid call which is ONHOLDPENDXFER. However, you *can* explicitly specify a different call you want to complete the transfer with. In CompleteConf, the same thing applies, but you *can* specify a different call or list of calls you want to conference.
<b>AgentRecording</b>	Required sub-message for AgentRecordingControl function. This allows the call system recording to be paused or unpaused. (Does not have any effect on user local call recording)

Function	Use for arg1
<b>BlindTransfer</b>	Transfer target
<b>Redirect</b>	Redirect target
<b>Dial</b>	Digit(s) to dial
<b>Park</b>	Parkslot
<b>SetupTransfer</b>	Optional target for assisted transfer call.
<b>AddToConf</b>	Conference target
<b>SetTag</b>	Tag text
<b>GenerateDigits</b>	DTMF digit(s) to play
<b>SetAccountCode</b>	Account code
<b>Set Auth code</b>	Auth code
<b>SetPriority</b>	'1', '2', or '3'

## Conference Member functions

message MemberFunctionData

```
{
  enum Action
  {
    None = 0;
    DropCall = 1;
    MuteOn = 2;
    MuteOff = 3;
  }
  int32 lref = 1;
  Action action = 2;
}
```

Use these functions to manage individual members of your conference. You will only be able to perform the action if you have sufficient privilege on the conference to do so.

### UnParkCall

**UnParkCall** is a payload to unpark a call

```
message UnParkCall
{
  string parkid = 1;
}
```

There is no guaranteed indication that an UnPark was successful, except that if successful, you will receive a CallInfo notification showing the unparked call (and the referencecallid supplied). You

would not normally expect to call this function unless you knew the parkid was occupied. You should subscribe to ParkHandler to keep track of parkslot occupancy.

If you specify a RequestID, you may get a useful fail code in the RequestResponse.

### *RecordingControls*

These apply to system recordings only. User can pause or Un-pause a call recording using RecordingControls.

```
Message
{
  subscribecmd
  {
    requestid=6
    subscribe_id=2
    callcontrol
    {
      callid=1
      callfunction
      {
        action=AgentRecordingControl
        agentrecording
        {
          pause ← Pause the recording
        }
      }
    }
  }
}
```

```
Message
{
  notify
  {
    subscribe_id=2
    callcontrol
    {
      refreshinstance=6
      callinfo
      {
        payload=PayloadFull
        callid=1
        state=CONNECTED
        direction=INBOUND
        gcid = 01 C0 A8 2A 7B 00 00 03 EA
        featuresavailable=0x29F09D DROP ...
        callingparty
        {
          number=0657765
        }
      }
    }
  }
}
```

```

connectedparty
{
  number=0657765
}
recordingpaused ← Recording now paused
}
}
}
}
}

```

## CallControl explicitly for Queue calls

If you have at least set “ccflags” bit 0 (0x01), you will receive call notifications whenever a call arrives in the queue, and a notification whenever that call changes. These call notifications will be in the form of a callinfo message.

You may receive multiple callinfo messages and multiple calllost messages in a single notify (when there are several calls), but for each call you will receive a maximum of one.

Several MTCTI3 clients can subscribe to the same queue, and they will all receive notifications. Note that if one client issues the “Finish” command (to end notifications), this will terminate the notifications on ALL clients.

Also note that before ‘Finish’, a call will only be reported on one Queue. If a call is in Sales, then is transferred to International, then the call will still be reported in Sales, and not in International, unless the application sends a Finish to stop the reporting in Sales.

This is because by default, the Queued call is reported through its entire lifetime.

If you only want to monitor the queued calls during the time they are queueing, you need to set “ccflags” bit 30 (0x40000000)

While the call is being handled by IP Office, the mtcti3 client will receive notifications about the call whenever the call information changes.

The client will only receive a Call Lost event once the call is completed unless the client explicitly sends a Finish event.

Additionally, the mtcti3 client has an Update capability to modify the call handling.

Lifetime of a simple call

CALLINFO	RINGING
CALLINFO	Targeting an agent
CALLINFO	Answered by agent
CALLINFO	Transferred to new Agent
CALLINFO	Agent has put call on hold

CALLLOST	Caller has hung up
----------	--------------------

### Actions on Queued calls

Actions can be performed on any call. Some actions simply enhance the queueing functionality which already exists in IP Office, and some actions completely override the default behaviour. For example, you can change the priority of a call, and the IP Office queueing mechanism will still be functioning. However, if you Redirect or Transfer the call, the queueing will be replaced.

If all you want to do is report what happens to a call that originally targets a queue, you do not need to perform any actions at all.

Actions that can be performed on a call are indicated in the featuresavailable bitfield. If you perform an action that is not available, the Action will be ignored.

Action	result
DropCall	Clears the call
ForceClear	If it is a regular call, it will clear the call. If it is a call into a meetme conference, it will terminate the conference.
SetTag	Changes the tag label on the call
SetAccountCode	Changes the account code of the call
SetPriority	Changes the call priority in the queue 1 = low 2 = medium 3 = high
Park	Parks the call to a parkslot
BlindTransfer	This is the major feature. You can use this to direct the call at any phase of its life.
CallRecordingOn/Off	Turns on/off call recording
AddToConf	When in a conference: Invite members to a conference
MemberFunction	When in a conference: Mute or drop conference members
Mute On/Off	When in a conference: Mute the caller
Finish	End the CTI association. The call will not end, but it will no longer generate CallInfo events.

The “powerful” function is:

BlindTransfer

### BlindTransfer action

Normally a BlindTransfer action would be used before the caller talks to an agent. You can use BlindTransfer to target an explicit agent, or to redirect the call to an explicit Queue, go to a pre-configured or interactive dialog with VMPro or to connect to an IP Office service like a MeetMe conference or an FNE.

BlindTransfer takes only one argument, but the “arg1” argument can be formatted to provide some extended functionality.

/ \$type/destination

Type can be:

102 = Voicemail

104 = Page

105 = Force autoanswer

106 = intrude

107 = Priority call

109 = CampOn

111 = Whisper

112 = Inclusion

116 = MeetMe conference

120 = FNE

BlindTransfer to a Callflow

So, to route to a particular VMPro callflow

```
Message
{
  subscribecmd
  {
    subscribe_id=7500
    callcontrol
    {
      callid=1
      callfunction
      {
        action=BlindTransfer
        arg1=/102/Callflow
      }
    }
  }
}
```

Transfers the call to “CallFlow” on VMPro

```
Message
{
  notify
  {
    subscribe_id=7500
    callcontrol
    {
      refreshinstance=2
    }
  }
}
```

```
    callinfo
    {
        callid=1
        state=ANSWEREDBYVOICEMAIL
        direction=INBOUND
        featuresavailable=0xE0E0E091 DROP BLINDXFER PARK TAG ACCT FORCECLEAR AUTH
REC+ PRIO FINISH
        calledparty=8010
        callingparty
        {
            number=03498984598
            nametype=9
        }
        connectedparty
        {
            number=Callflow
        }
        trunkdetail
        {
            trunktype=TrunkISDN
            did=8010
        }
        queuedetail
        {
            priority=1
        }
    }
}
}
```

## BlindTransfer to a MeetMe Conference

```
Message
{
    subscribecmd
    {
        subscribe_id=7500
        callcontrol
        {
            callid=1
            callfunction
            {
                action=BlindTransfer
                arg1=/116/20987
            }
        }
    }
}
```

## Result

```
Message
{
  notify
  {
    subscribe_id=7500
    callcontrol
    {
      refreshinstance=3
      callinfo
      {
        callid=1
        state=CONNECTED
        direction=INBOUND
        featuresavailable=0x60E0F801 DROP ADDTOCONF CONFMEMBER TAG ACCT
        FORCECLEAR AUTH REC+ PRIO
        calledparty=8010
        callingparty
        {
          number=03498984598
          nametype=110
        }
        connectedparty
        {
          number=20987
          name=Conf 20987
          nametype=110
        }
        trunkdetail
        {
          trunktype=TrunkISDN
          did=8010
        }
        queuedetail
        {
          priority=1
        }
      }
    }
  }
}
```

You see the call is now connected to the conference.

## BlindTransfer to a conference requiring PIN access

```
Message
{
  subscribecmd
  {
    subscribe_id=7500
```

```
callcontrol
{
  callid=1
  callfunction
  {
    action=BlindTransfer
    arg1=/116/2005;MODEFROMPIN(123456)
  }
}
}
```

Where the PIN is 123456

The character between the 5 and the M is a semi-colon.

### BlindTransfer to a functional queue

You can transfer a call from the “owner” queue to another IP Office queue with a simple BlindTransfer to the new queue. This call will continue to be monitored here as it is handled by the other queue, and you can abandon the queue at any time by performing another BlindTransfer.

```
Message
{
  subscribecmd
  {
    subscribe_id=2
    callcontrol
    {
      callid=2
      callfunction
      {
        action=BlindTransfer
        arg1=2502
      }
    }
  }
}
```

(2502 is a huntgroup)

```
Message
{
  notify
  {
    subscribe_id=2
    callcontrol
```

```

{
  callinfo
  {
    callid=3
    state=QUEUEING
    direction=INBOUND
    featuresavailable=0x6060E091 DROP BLINDXFER PARK TAG ACCT FORCECLEAR AUTH
  }
  calledparty=8010
  callingparty
  {
    number=03498984598
    nametype=9
  }
  trunkdetail
  {
    trunktype=TrunkISDN
    did=8010
  }
  queuedetail
  {
    priority=1
    overflownumber=2502
    overflowname=Service Queue
  }
  targets
  {
    target
    {
      partyinfo
      {
        number=2005
        name=Bergcamp
        nametype=5
      }
    }
  }
}

```

You will continue to get notify events as the queue changes the agents that are targeted, and after the call is answered.

## Error codes

Error	Code	Description
MTCTISESS_SUCCESS	0	Success
MTCTISESS_ERRUNKNOWN	1	Unknown error
MTCTISESS_UNPACKERR	2	Message unpack error
MTCTISESS_NOTINSTRUMENTED	3	Not instrumented
MTCTISESS_NOTFOUND	4	Not found
MTCTISESS_TOOMANY	5	Too many
MTCTISESS_TOOBIG	6	
MTCTISESS_USERNOTFOUND	7	
MTCTISESS_SERVICE_NOT_AVAILABLE	8	
MTCTISESS_NOTALLOWED	9	
MTCTISESS_SUBSCRIPTION_INVALID	100	Not a recognized subscription
MTCTISESS_SUBSCRIPTION_INVALID_ID	101	Subscription Update with invalid ID
MTCTISESS_SUBSCRIPTION_TIMEOUT_TOO_SMALL	102	
MTCTISESS_GENCMD_ERRUNKNOWN	150	General Command Error
MTCTISESS_GENCMD_INVALID_PAYLOAD	151	General Command badly formatted
MTCTISESS_GENCMD_REQUIRED_DATA_MISSING	152	General Command with mandatory element missing
MTCTISESS_PRESENCE_ERRUNKNOWN	500	Any error to do with presence subscription
MTCTISESS_REQUESTFAILED	5000	A valid command has failed
MTCTISESS_REQUESTTIMEOUT	5001	A valid command has taken too long to create a response
MTCTISESS_REQUEST_INVALID_PAYLOAD	5002	Missing or field out of range.
MTCTISESS_REQUEST_INVALID_CONTEXT	5003	Can find context for this action

#### Specific Error codes for Call Control

In some case IP Office can be more specific about the reason why an Action failed. If it can't be more specific, it will return one of the generic Error codes above:

MTCTISESS_CALLCONTROL_NOPHONE	600	Cant make a call, or unpark a call because the user is not logged in to any handset
MTCTISESS_CALLCONTROL_EXTNFAULT	601	Cant Make a call because the user's phone is not connected
MTCTISESS_CALLCONTROL_CALLNOTFOUND	602	Can perform action on this call, because the call cannot be found
MTCTISESS_CALLCONTROL_MAXCALLS	603	Cant Make a call because the user has no more call appearances.
MTCTISESS_CALLCONTROL_BADACCT	604	The account code entered is not valid
MTCTISESS_CALLCONTROL_BDAUTH	605	The authorization code is not valid
MTCTISESS_CALLCONTROL_TARGETNOTFOUND	606	Typically for UnPark, or AddToConference, cannot find the thing you are targeting.
MTCTISESS_CALLCONTROL_PERMISSION	607	You do not have permission to perform this action
MTCTISESS_CALLCONTROL_BADFORMATTING	608	One of the fields is missing or string is too long or number is out of range
MTCTISESS_CALLCONTROL_INVALIDCALLSTATE	609	You can't do this action at this time.
MTCTISESS_CALLCONTROL_CANTBEDONE	610	Typically you cannot answer the call on this phone because you need to physically pick up the handset.

MTCTISESS_CALLCONTROL_NOCOVERAGE	611	You cant Drop a ringing call if there is no coverage destination.
MTCTISESS_CALLCONTROL_TRANSFERFAILED	612	The blind transfer target is invalid or refused the call.
MTCTISESS_CALLCONTROL_PARKFAILED	613	Could not park this call
MTCTISESS_CALLCONTROL_OTHERNOTFOUND	614	Trying to CompleteTransfer or CompleteConf with invalid related calls.
MTCTISESS_CALLCONTROL_CANTCOMPLETE	615	TransferComplete not allowed, maybe because of the nature of the calls you are trying to join.
MTCTISESS_CALLCONTROL_UNSUPPORTED	616	Not a supported function
MTCTISESS_CALLCONTROL_ALREADYDONE	617	The command would have no effect

## Features Available

Meaning of “FeaturesAvailable” in CallInfo

Bit 0	Drop
Bit 1	Answer Call
Bit 2	Hold call
Bit 3	UnHold call
Bit 4	Blind Transfer
Bit 5	Redirect
Bit 6	Dial
Bit 7	Park
Bit 8	SetupTransfer
Bit 9	CompleteTransfer
Bit 10	CompleteConf
Bit 11	AddToConf
Bit 12	AdminConfMember
Bit 13	SetTag
Bit 14	SetAccountCode
Bit 15	reserved
Bit 16	PushToEC500
Bit 17	GenerateDigits
Bit 18	reserved
Bit 19	reserved
Bit 20	RecordingPauseControl
Bit 21	ForceClear
Bit 22	SetAuthCode
Bit 23	CallRecordingOn
Bit 24	CallRecordingOff
Bit 25	PrivacyOn
Bit 26	PrivacyOff
Bit 27	MuteOn
Bit 28	MuteOff
Bit 29	SetPriority
Bit 30	Finish

Call functions individually described

### DropCall

TAPI equivalent: lineDrop()

Control: only try to do this if bit0 of “featuresavailable” is set.

### **Arg1**

not used

### **Line types**

User or Queue or ParkServer

### **Action**

User: IP Office will try to clear the call from this user. If call is ringing, it will try to send the call to coverage. If call is answered, the call will be cleared. If connected to a conference, the user will be dropped out of the conference. This does not necessarily clear the conference.

Queue or Park Server: Call will be dropped.

### **Errors**

DropCall may fail even if bit0 is set.

If it fails, and you have populated the 'requestid' field, you will get the error in the RequestResponse:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_NOTFOUND

If it fails due to (eg) an inability to find a suitable coverage target, you may get MTCTISESS\_SUCCESS but the call will not drop. In this case you will usually get MTCTISESS\_CALLCONTROL\_NOCOVERAGE. This is a good example of the rule that the application should only use Notify updates to observe what is actually going on.

### **AnswerCall**

TAPI equivalent: lineAnswer()

Control: only try to do this if bit1 of "featuresavailable" is set.

### **Arg1**

not used

### **Line types**

User only

### **Action**

IP Office will try to answer a ringing call at this user. If the user has multiple simultaneous devices ringing at the same time for the same call, then IP Office will choose the most appropriate device to answer the call. This is chosen in the order:

Desk phone or teleworker

Soft phone

Mobile Equinox application

It is possible to be more precise by using the 'devicehint' in the UpdateCallControl payload. If you specify SOFTPHONEANY it will only answer the call on a softphone.

For some phone types, like a ringing POTS phone, it is not possible to Answer a call through CTI. (CTI cannot take the phone off-hook). Generally in this case, bit1 of featuresavailable should be unset.

Also, note that if there is already a Connected call, Answering a ringing call may result either in the Answered call being Answered-to-OnHold, or the previously connected call to be demoted to OnHold.

### **Errors**

AnswerCall may fail even if bit1 is set.

If it fails, and you have populated the 'requestid' field, you will get the error in the RequestResponse:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Hard pots phone on hook: MTCTISESS\_CALLCONTROL\_CANTBEDONE

Unsuitable action (eg if call was already answered manually while the command was in transit):

MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

### **HoldCall**

TAPI equivalent: lineHold()

Control: only try to do this if bit2 of "featuresavailable" is set.

### **Arg1**

not used

### **Line types**

User only

### **Action**

IP Office will try to put an active call on Hold. For the person talking to this user, he may expect to hear HoldMusic.

Note.

When putting a call onHold, the IP Office may initiate a Hold-reminder timer and after the Hold-reminder expires a deskphone may start ringing. None of this is reflected in CTI3. The OnHold call stays onHold even while the deskphone is doing Ring-reminder, and can only go back to Connected using UnHold (not Answer)

### **Errors**

HoldCall may fail even if bit2 is set.

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Already on Hold: MTCTISESS\_CALLCONTROL\_ALREADYDONE

Call not in a state where it can be put on Hold: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

## UnHoldCall

TAPI equivalent: lineUnHold()

Control: only try to do this if bit3 of "featuresavailable" is set.

### Arg1

not used

### Line types

User only

### Action

IP Office will try to make a previously Held Call to Connected state. This would generally automatically force any other Connected call into Held.

### Errors

UnHoldCall may fail even if bit3 is set.

Some 3<sup>rd</sup> party SIP handsets cannot be coaxed through CTI to take an OnHold call and UnHold it.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Already Active: MTCTISESS\_CALLCONTROL\_ALREADYDONE

Call not in a state where it can be taken OffHold: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

Can't be done on this type of phone: MTCTISESS\_CALLCONTROL\_CANTBEDONE

## BlindTransfer

TAPI equivalent: lineBlindTransfer()

Control: only try to do this if bit4 of "featuresavailable" is set.

### Arg1

The transfer-to destination. In a successful BlindTransfer, the connected or ringing call is immediately disconnected from the user and is sent to another destination specified by arg1. BlindTransfer frequently fails if the transfer-to destination is not a valid target. The minimum length of the string is 1, and the maximum length is 78.

Characters in this field would normally be 0-9, '#' '\*' but for exotic transfers, other characters may be expected. Non-ascii characters must be encoded in Utf8.

### **Line types**

User or Queue or ParkServer

### **Action**

IP Office will try transfer the call to the specified destination.

### **Errors**

BlindTransfer may fail even if bit4 is set.

If the transfer does not succeed, the call will stay with the user.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

cannot redirect the call: MTCTISESS\_CALLCONTROL\_TRANSFERFAILED

### **Redirect**

TAPI equivalent: lineRedirect()

Control: only try to do this if bit5 of "featuresavailable" is set.

### **Arg1**

The transfer-to destination. In a successful Redirect, the ringing call is immediately disconnected from the user and is sent to another destination specified by arg1.

Redirect frequently fails if the transfer-to destination is not a valid target.

The minimum length of the string is 1, and the maximum length is 78.

Characters in this field would normally be 0-9, '#' '\*' but for exotic transfers, other characters may be expected. Non-ascii characters must be encoded in Utf8.

### **Action**

IP Office will try to redirect the call to the specified destination.

### **Line types**

User or Queue

### **Errors**

Redirect may fail even if bit5 is set.

If the redirect does not succeed, the call will stay with the user.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

cannot redirect the call: MTCTISESS\_CALLCONTROL\_TRANSFERFAILED

### **Dial**

TAPI equivalent: lineDial()

Control: only try to do this if bit6 of "featuresavailable" is set.

### **Arg1**

The minimum length of the string is 1, and the maximum length is 78. But it would be unexpected for this to be anything other than 1 as this is used primarily for overlap dialling.

Characters in this field would normally be 0-9, '#' '\*'

When dialling a destination, IP Office will append the supplied digits to the dialled string.

### **Line types**

User only

### **Action**

When using overlap dialling, IP Office will progress a call from Dialtone to Dialling to Dialed as the target number is resolved.

### **Errors**

Dial may fail even if bit6 is set.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

## Park

TAPI equivalent: linePark()

Control: only try to do this if bit7 of “featuresavailable” is set.

### **Arg1**

The minimum length of the string is 1, and the maximum length is 9.

Characters in this field would normally be 0-9, ‘#’ ‘\*’

### **Line types**

User or Queue

### **Action**

There is some risk to trying to park a call if you have no knowledge of the state of the park slot you are trying to use. For this reason, it would be recommended that a private park slot is used, or the application has a subscription to the PARKSERVER so it knows which park slots are free.

### **Errors**

Park may fail even if bit7 is set.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can’t find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

## SetupTransfer

TAPI equivalent: lineSetupTransfer()

Control: only try to do this if bit8 of “featuresavailable” is set.

### **Arg1**

This field is optional. If you do not include it, the new call created will go into the DIALTONE state. If it is included, the new call will use this string as the target to dial. This string has to be a complete number. You will not have the opportunity to dial further.

The minimum length of the string is 0, and the maximum length is 78

Characters in this field would normally be 0-9, ‘#’ ‘\*’ but for exotic transfers, other characters may be expected. Non-ascii characters must be encoded in Utf8.

### **Line types**

User only

### **Action**

When performing SetupTransfer on a call, you are creating a new call (the assistant transfer call) which is related to the original call. The original call should transition to HOLDFORTRANSFER state and show “relatedcallid” association with this new call.

You cannot specify the “referencecallid” for this new call.

### **Errors**

SetupTransfer may fail even if bit8 is set.

If a destination is specified and the target is invalid, this function should succeed, and a new call is created with state == FAILED.

This would fail if the phone is a digital phone with no spare call appearances.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can’t find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

No spare call appearances: MTCTISESS\_CALLCONTROL\_MAXCALLS

### **SetupConf**

TAPI equivalent: lineSetupConference()

Control: only try to do this if bit8 of “featuresavailable” is set (same bit as SetupTransfer).

### **Arg1**

This field is optional. If you do not include it, the new call created will go into the DIALTONE state. If it is included, the new call will use this string as the target to dial. This string has to be a complete number. You will not have the opportunity to dial further.

The minimum length of the string is 0, and the maximum length is 78

Characters in this field would normally be 0-9, ‘#’ ‘\*’ but for exotic transfers, other characters may be expected. Non-ascii characters must be encoded in Utf8.

### **Line types**

User only

### **Action**

This function is practically identical to SetupTransfer

When performing SetupConf on a call, you are creating a new call (the assistant transfer call) which is related to the original call. The original call should transition to HOLDFORCONF state and show “relatedcallid” association with this new call.

You cannot specify the “referencecallid” for this new call.

## **Errors**

SetupConf may fail even if bit8 is set.

If a destination is specified and the target is invalid, this function should succeed, and a new call is created with state == FAILED.

This would fail if the phone is a digital phone with no spare call appearances.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

No spare call appearances: MTCTISESS\_CALLCONTROL\_MAXCALLS

## **CompleteTransfer**

TAPI equivalent: lineCompleteTransfer( LINETRANSFERMODE\_TRANSFER)

Control: only try to do this if bit9 of "featuresavailable" is set.

## **Arg1**

Not used

## **Callinst**

This may be used, in which case it would override any "relatedcallid" relationship and may transfer together two calls which were previously unrelated. Only 0 or 1 callinst should be specified.

## **Line types**

User only

## **Action**

CompleteTransfer joins two calls together and drops the user out of the call.

CompleteTransfer can be called without callinst, as long as there is a related\_callid. If there is no callinst, and no related\_callid, the Completion of the transfer will fail.

## **Errors**

CompleteTransfer may fail even if bit9 is set.

There are several reasons why a CompleteTransfer may fail. There may not be two calls to join together. The two calls specified may not be allowed to be joined together (eg two public calls may not be allowed to talk together without an internal party, or joining the two calls together may result in a call which cannot be cleared).

It is not always possible for the MTCTI3 application to know in advance whether the transfer will succeed or fail.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

Cant find call inst: MTCTISESS\_CALLCONTROL\_OTHERNOTFOUND

The transfer targets are incompatible: MTCTISESS\_CALLCONTROL\_CANTCOMPLETE

## CompleteConference

TAPI equivalent: lineCompleteTransfer( LINETRANSFERMODE\_CONFERENCE)

Control: only try to do this if bit10 of "featuresavailable" is set.

### Arg1

Not used

### Callinst

This may be used, in which case it would override any "relatedcallid" relationship and may conference together two calls which were previously unrelated.

There can be several calls listed in the callinst list in which case all the listed calls will be joined to the conference

### Line types

User only

### Action

CompleteConference joins two or more calls together into a conference.

CompleteConference can be called without callinst, as long as there is a related\_callid. If there is no callinst, and no related\_callid, the Completion of the conference will fail.

When one of the calls is already a conference, the other call will be joined into the conference.

There are many rules about joining parties into a conference. Some parties are not allowed to join some conferences, or the conference capacity may be reached.

This function either fully succeeds or fully fails. If any party is not allowed into the conference, then no parties will join.

### Errors

CompleteConference may fail even if bit10 is set.

It is not always possible for the MTCTI3 application to know in advance whether the CompleteConference will succeed or fail. MTCTI3 will not give a useful reason for failing the function and will not identify any rogue call which is blocking the function completion.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

Can't find call inst: MTCTISESS\_CALLCONTROL\_OTHERNOTFOUND

The conference targets are incompatible: MTCTISESS\_CALLCONTROL\_CANTCOMPLETE

There is a privacy issue in making this conference: MTCTISESS\_CALLCONTROL\_PERMISSION

## AddToConference

TAPI equivalent: lineAddToConference()

Control: only try to do this if bit11 of "featuresavailable" is set.

### Arg1

This must be a string of length between 1 and 78 digits. It is the target address of the invited party to the conference. It must be a complete number

### Line types

User or Queue

### Action

This is only allowed if the call is already connected to a conference, and the user has privileges in that conference to invite new conference members.

### Errors

AddToConference may fail even if bit11 is set.

If the number which is used to dial the new conference member is invalid, then normally there is a new ConferenceMember created, with state=FAILED. This member call will then have to be dropped.

The function AddToConference reports SUCCESS in this case.

AddToConference may fail if there are insufficient conference resources, or the conference capacity is reached, or the user does not have the privilege to perform the function.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE  
There is a permission error: MTCTISESS\_CALLCONTROL\_PERMISSION  
arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING  
Could not target the destination: MTCTISESS\_CALLCONTROL\_TARGETNOTFOUND

## MemberFunction

TAPI equivalent: lineDrop() – for action=Drop, none for mute/unmute

Control: only try to do this if bit12 of “featuresavailable” is set.

### Arg1

Not used

### Line types

User or Queue

### Memberfunctiondata

This is required. It specifies which conference member you wish to Mute/UnMute/Drop. It also specifies which of these three functions is to be performed.

### Action

This is only allowed if the user has the privilege to perform these functions in this conference.

### Errors

MemberFunction may fail even if bit12 is set.

If the referred to conference member does not exist.

If the user does not have the privilege.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED  
Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT  
Can’t find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND  
Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE  
Can’t find lref: MTCTISESS\_CALLCONTROL\_OTHERNOTFOUND  
There is a permission error: MTCTISESS\_CALLCONTROL\_PERMISSION  
Unknown command: MTCTISESS\_CALLCONTROL\_UNSUPPORTED

## SetTag

TAPI equivalent: lineSetCallData()

Control: only try to do this if bit13 of “featuresavailable” is set.

### **Arg1**

Can be a string of length 0 – 127 Unicode characters after converting from Utf8 to BMP-0. If there are NULL characters in the callData, the tag is effectively truncated at the NULL.

If empty, this clears the call tag.

### **Line types**

User or Queue or Parkserver

### **Action**

Adds a call label to the call, which is distributed with the call if it is transferred.

### **Errors**

This function does not fail on a valid call. If the string length is more than 127 characters, it will be truncated.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

## SetAccountCode

TAPI equivalent: lineCallDevSpecific( )

Control: only try to do this if bit14 of “featuresavailable” is set.

### **Arg1**

Can be a string of length 0 – 15 Unicode characters after converting from Utf8 to BMP-0.

If empty, this clears the account code.

### **Line Types:**

User or Queue

### **Action**

Tags the call with the specified account code.

Only pre-configured account codes are allowed to be entered, unless there are wild-card account codes in the IP Office config.

### **Note**

If the MTCTI3 application wants to know the list of account codes configured on IP Office, there is a pseudo-file that can be read using the GeneralCmd "GetFile". The file to read is "nasystem/AccountCode"

### **Errors**

This function will fail if the account code is not a valid code matching one in the IP Office configuration.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

Not a recognized account code: MTCTISESS\_CALLCONTROL\_BADACCT

### **SetNotes**

Not supported in this release

### **PushToEC500**

TAPI equivalent: none

Control: only try to do this if bit16 of "featuresavailable" is set.

### **Arg1**

Not used.

### **Line Types**

User only

### **Action**

This only works for users who have a Mobile Twinning destination configured. This function starts the process of transferring the call to the mobile twin device, but the transfer only completes if the call is answered on the mobile. While the mobile is still ringing, the caller can still talk to the user.

### **Errors**

This function will fail if Mobile twinning destination is not set up or cannot be targeted. This function returns SUCCESS once the push is initiated. It does not wait until the transfer completes before it reports the result.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE  
Could not target mobile: MTCTISESS\_CALLCONTROL\_TRANSFERFAILED

### [GenerateDigits](#)

TAPI equivalent: lineGenerateDigits( )

Control: only try to do this if bit17 of “featuresavailable” is set.

#### **Arg1**

Required. The length should be in the range 1 – 32 characters 0-9, ‘\*’ ‘#’

#### **Line Types**

User only

#### **Action**

This sends DTMF to the far end of the call. Each character in the string is sent individually.

#### **Errors**

This function will succeed if any character in the string is sent. It may be that the call is dropped part way through the generate digit string, in which case the result is still SUCCESS.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can’t find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 missing or invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

### [ShortCodeAction](#)

Not supported in this release

### [AnswerPage](#)

TAPI equivalent: none

Control: only try to do this if bit0 of “featuresavailable2” is set.

#### **Arg1**

Not used

#### **Line Types**

User only

### **Action**

If this call is an inbound Page call (so you are hearing a Page) you can convert this to a 2-way conversation using this function.

The “pagecall” field in the Callinfo will indicate that it is an incoming page call.

### **Errors**

This function will succeed only if the call is an incoming Page call, and the user is allowed to convert the call.

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

permission error: MTCTISESS\_CALLCONTROL\_PERMISSION

ForceClear

TAPI equivalent: none

Control: only try to do this if bit21 of “featuresavailable” is set.

### **Arg1**

Not used

### **Line Types**

User or Queue or Parkserver

### **Action**

This is a brutal function and should not normally be offered. If the user is in a conference, the conference will be terminated.

If the user is receiving an incoming call, the call will be cleared all the way to the source. You should normally use Drop.

### **Errors**

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

### SetAuthCode

TAPI equivalent: lineCallDevSpecific( )

Control: only try to do this if bit22 of "featuresavailable" is set.

#### **Arg1**

Can be a string of length 0 – 15 Unicode characters after converting from Utf8 to BMP-0.

If empty, this clears the auth code.

#### **Line Types**

User only

#### **Action**

Tags the call with the specified auth code.

Only pre-configured auth codes are allowed to be entered.

#### **Note**

Auth codes are permissions, so if you enter a valid auth code, you are allowed to make certain calls. An Auth code is generally associated with a user, so provides executive users with more permissions. The user who owns the code is nominally billed for the call.

The MTCTI3 application does not have access to a list of valid auth codes.

#### **Errors**

This function will fail if the auth code is not a valid code matching one in the IP Office configuration.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

Not a recognized auth code: MTCTISESS\_CALLCONTROL\_BDAUTH

### CallRecordingOn/Off

TAPI equivalent: lineCallDevSpecific( )

Control: only try to do this if bit23/24 of "featuresavailable" is set.

#### **Arg1**

Not used

### **Line Types**

User only

### **Action**

User: Starts or stops personal call recording of the call (to the user's mailbox).

### **Errors**

This function will fail if the call is private, or the voicemail does not have the functionality or capacity.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

permission error: MTCTISESS\_CALLCONTROL\_PERMISSION

### **PrivacyOn/Off**

TAPI equivalent: lineCallDevSpecific( )

Control: only try to do this if bit25/26 of "featuresavailable" is set.

### **Arg1**

Not used

### **Line Types**

User only

### **Action**

Makes a call locally private. This will prevent call recording of the call you are on, as long as you have enough authority. (If you are a minor delegate in a big conference, you cannot stop the conference from being recorded)

Making a call not-private only means you have unset your own privacy. If another party to the call has set their own privacy, you cannot override that.

### **Errors**

Generally succeeds.

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

### MuteOn/Off

TAPI equivalent: none

Control: only try to do this if bit27/28 of “featuresavailable” is set.

### Arg1

Not used

### Line Types

User only

### Action

This only applies if this user is in a conference and wishes to manipulate his own mute status in the conference.

It does not change the Mute setting on his handset/headset. This is just because you cannot control this using CTI on most phones.

### Errors

If it fails and you have populated the ‘requestid’ field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can’t find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

Not possible to perform this function: MTCTISESS\_CALLCONTROL\_CANTBEDONE

### SetPriority

TAPI equivalent: none

Control: only try to do this if bit29 of “featuresavailable” is set.

### Arg1

A text string denoting the new priority. 1 = low priority, 2 = medium priority, 3 = high priority.

### Line Types

Queue only

### Action

This changes the priority of the call in the current queue. It does not persist after a call is answered and transferred to a new queue.

There are only 3 allowed values of Arg1 : “1”, “2”, or “3”

In general, calls with a higher priority are answered first in a queueing situation.

### **Errors**

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

Not possible to perform this function: MTCTISESS\_CALLCONTROL\_CANTCOMPLETE

### **Finish**

TAPI equivalent: none

Control: only try to do this if bit30 of "featuresavailable" is set.

### **Arg1**

If included, must be "1".

### **Line Types**

Queue only

### **Action**

This disassociates the call from this Queue. If you do not disassociate the call, it will continue to be followed on this queue until the call ends. You 'Finish' the call if your application has no further interest in this call.

MTCTI3 only follows a call on one Queue at a time, so if you want to view it on a new Queue, you need to "Finish" it on the old Queue. Using Arg1 = "1" means that it will immediately start reporting on any new queue it is associated with. Otherwise it only starts reporting when it subsequently arrives at the new Queue.

### **Errors**

If it fails and you have populated the 'requestid' field, you will get the error:

Unspecified Error: MTCTISESS\_REQUESTFAILED

Remote SCN target not responding: MTCTISESS\_REQUESTTIMEOUT

Can't find CallID: MTCTISESS\_CALLCONTROL\_CALLNOTFOUND

Call not in a state to perform this function: MTCTISESS\_CALLCONTROL\_INVALIDCALLSTATE

arg1 invalid: MTCTISESS\_CALLCONTROL\_BADFORMATTING

Not possible to perform this function: MTCTISESS\_CALLCONTROL\_CANTCOMPLETE

## Alternative connection methods

### Noframing

If you do not want to handle the protocol framing (1<sup>st</sup> 4 octets of each message contains {0,0,0,1} header), you can instead use the websocket protocol “tpkt/openapi\_noframing”

(that is “noframing” appended to the end of the string)

Then the message stream in both directions will be pure protocol buffers.

## Limits

*A maximum of 10 x MTCTI3 connections are allowed per IPOffice.*

*The maximum number of presentites subscribed per MTCTI 3 connection is the same as the user limit on the system..*

## Version Compatibility

This is supported on IPOffice 11.1.0.0 and future versions until such time as it is withdrawn. In Release 11.0.4.2, there is some support, but some error reporting may be missing.

## Resilient solutions

For a resilient solution with an IP Office primary and an IP Office secondary, the client MTCTI3 application can work in a live-live deployment. Connections to either PBX will render more-or-less identical information for users and commands can be sent down either connection (we do not commit to a particular level of identity). For queues, the configuration and command interfaces are mirrored, but the view of the queued calls differs on each system.

Calls queueing on the primary can be seen and controlled on the primary

Calls queueing on the secondary can be seen and controlled on the secondary.

### What this means

In an IP Office failover, you would want the IP Office secondary to act as a resilient backup of the IP Office primary for groups. Then the queues on the secondary automatically become active when the primary is down.

### Resilient app

As the IP Office allows for up to 10 x MTCTI3 connections, the client application can itself have a resilient live-live twin.

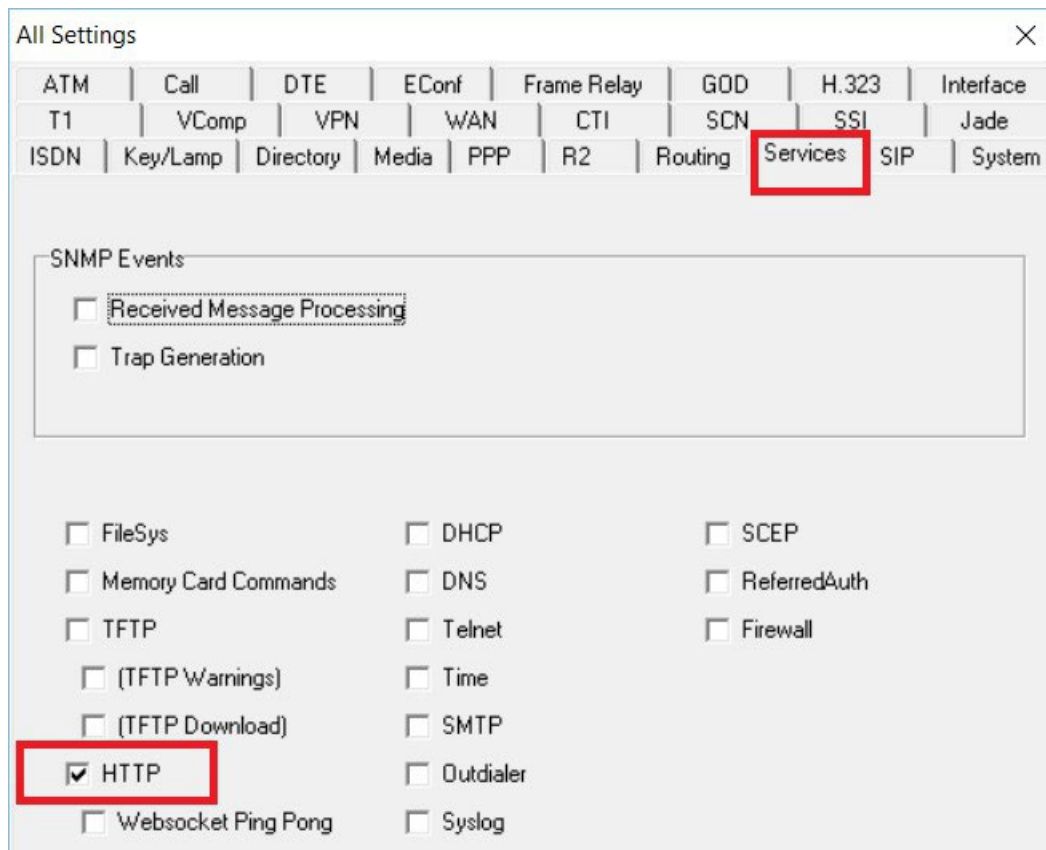
## Development tools

The most important development tool will be SysMonitor. SysMonitor will decode all the messages that your application sends and receives.

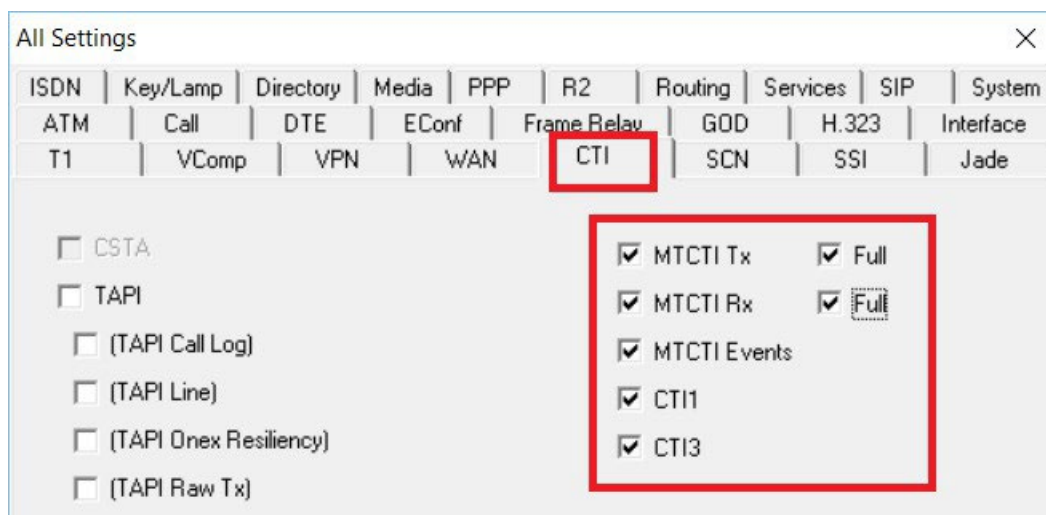
## SysMonitor

When using SysMonitor, you should enable the following flags to decode CTI3 connections and protocols.

### For Web sockets



### For CTI3 protocols



When a line is open, you can perform actions on the line.

## Getting started with the proto file

A good starting point is to download the latest protobuf code from github. At time of writing, the latest version is v3.10.0.

On github, there are protobuf files for: c++, c#, java, js, objective, php, python, and ruby. (example protobuf-cpp-3.10.0.zip)

There is also a version of the 'proto' file compiler called 'protoc'. You should match the same version of 'protoc' compiler with the protobuf source code for a successful compile.

You need these steps to build and decode messages automatically. Now it is perfectly possible to write your own code to encode and decode the messaging as the protocol buffer encoding technique is published by google, but this would not be recommended because of the ready availability of these tools and implementations.

### C++ and visual studio

Note that for C++ the v3.10.0 protobuf code requires a C++ 11 compiler. On visual studio, this is vs2017 or later. For a version that does not require a C++ 11 compiler, you need to go back to v3.5.0 or earlier. These earlier versions work perfectly well with our ipo\_mtcti3.proto file, but there are speed optimizations that may be available with the later protobuf code.

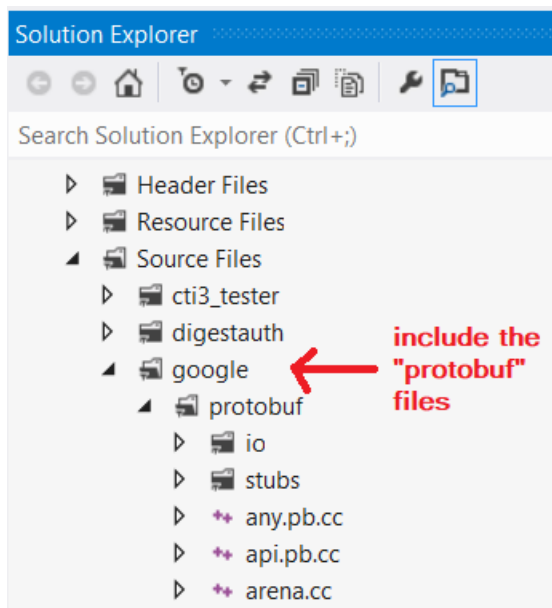
*Converting "ipo\_mtcti3.proto" into source code:*

```
R:\google>protoc --cpp_out=R:\google ipo_mtcti3.proto
```

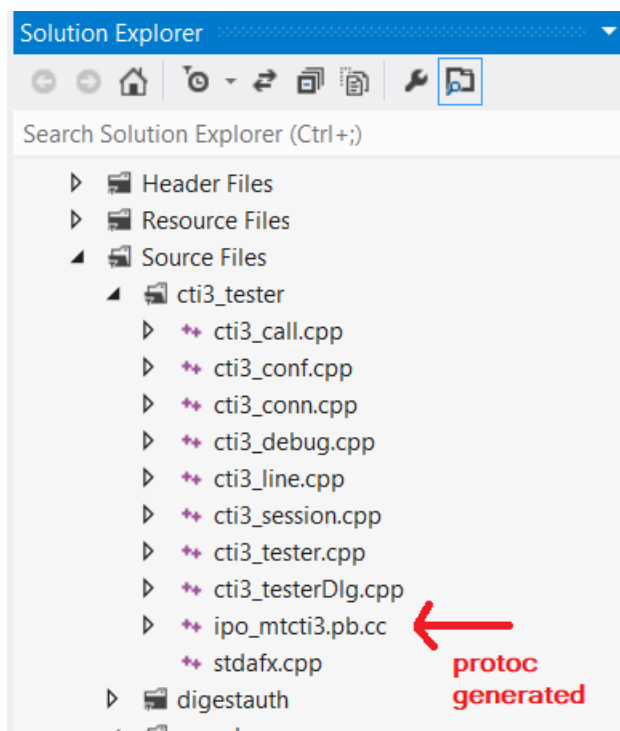
Produces:

**Directory of R:\google**

```
24/10/2019 09:20 <DIR>      .
24/10/2019 09:20 <DIR>      ..
24/10/2019 09:20          1,034,570 ipo_mtcti3.pb.cc ← generated
24/10/2019 09:20          665,490 ipo_mtcti3.pb.h  ← generated
09/10/2019 11:26          13,726 ipo_mtcti3.proto
22/10/2019 16:22 <DIR>      protobuf ← unzipped
22/10/2019 15:38          5,281,431 protobuf-cpp-3.5.0.zip ← github
22/10/2019 15:38          1,256,007 protoc-3.5.0-win32.zip ← github
22/10/2019 15:41          4,029,440 protoc.exe ← unzipped
          6 File(s)          12,280,664 bytes
          4 Dir(s)  45,700,952,064 bytes free
```



Also include



When you have connected your application you are ready to send and receive protocol buffers.

They are encoded in C++ like this:

```
#include <mtcti/ipo_mtcti3.pb.h>
#include <string>
void CTI3Session::FrameAndTransmit( std::string * obuf )
```

```

{
    int x = obuf->length( );
    UBYTE * dp = new UBYTE[x+4];
    memcpy( dp+4, obuf->data( ), x );
    dp[0] = 0;
    dp[1] = 0;
    dp[2] = 0;
    dp[3] = 1;
    TxFramedMessage( &dp[0], x + 4 );
    delete[] dp;
}

void CTI3Session::StartCTI3Subscribe( CTI3Lines * Alines )
{
    Message msg;
    std::string obuf;
    Subscribe * s = msg.mutable_subscribe( );
    s->set_requestid( nextrequestid++ );
    s->set_subscribe_id( Alines->subscriptionid );
    s->set_timeout( Alines->timeout );
    SubscribeLines * slines = s->mutable_lines( );
    slines->set_flags( 7 );
    msg.SerializeToString( &obuf );
    FrameAndTransmit( &obuf );
}

```

And decoding:

```

void CTI3Session::RxFramedMessage( UBYTE * dp, int len )
{
    if( len > 4 )
    {
        if( (dp[0] == 0) && (dp[1] == 0) && (dp[2] == 0) && (dp[3] == 1) )
        {
            std::string istring( dp + 4, dp + len - 4 );
            Message m;
            m.ParseFromString( istring );
            if( m.has_notify() )
            {
                ULONG subscribeid = (ULONG )m.notify().subscribe_id();
                ULONG notifyid = (ULONG )m.notify().notify_id();
                CTI3GeneralSubscription * ss = FindSubscription( subscribeid );
                if( ss )
                {
                    if( ss->OnNotify( m.notify() ) )
                    {
                        NotifyAck( subscribeid, notifyid );
                    }
                }
            }
        }
    }
}

```

```
// Add your code here for other payloads
    }
  }
}
```

## Java

You get “protobuf-java-3.10.0.jar” from github

Compiling the proto file using protoc yields “IpoMtcti3.java”

```
R:\google>protoc --java_out=R:\google ipo_mtcti3.proto
...
24/10/2019 17:04          2,099,987 IpoMtcti3.java
```

These two objects (the jar and the java) go together.

The primary object is “IpoMtcti3.Message”

To build an serialize a simple lines subscription, looks something like:

```
IpoMtcti3.SubscribeLines linesSubscribe =
IpoMtcti3.SubscribeLines.newBuilder().setFlags(1).build();
IpoMtcti3.Subscribe subscribeMsg =
IpoMtcti3.Subscribe.newBuilder().setSubscribeId(subscribeId)
    .setRequestId(26)
    .setTimeout(0)
    .setLines(linesSubscribe)
    .build();
IpoMtcti3.Message Msg =
IpoMtcti3.Message.newBuilder().setSubscribe(subscribeMsg).build();

target.sendProtoMsg(Msg.toByteArray()); // Need to prepend the framing...
```

To decode messages from the line:

```
public void handleMessage(byte[] message) {
    byte[] msgBytes = source.afterReceive(message);
    try {
        IpoMtcti3.Message Msg = IpoMtcti3.Message.parseFrom(msgBytes);
        clientEndPoint.processMessageFromIPO(Msg);
    } catch (InvalidProtocolBufferException ipbe) {
        System.out.println("Invalid protocol buffer exception");
    }
}
```

## Javascript

```
R:\google>protoc --js_out=R:\google ipo_mtcti3.proto
```

This generates a bunch of js files for each defined object, the main one being:

Message.js

This has the functions to serialize and deserialize the binary data into and out of the ‘Message’ object

I don’t have any code for using this.

## Establishing a Websocket connection

```
HTTP: 192.168.42.31(4096)-(443) HTTPSession(Secure) (Total = 2)
HTTP: 192.168.42.31(4096)-(443) HTTPSession: Operational
```

```

HTTP: 192.168.42.31(4096)-(443) HTTPSession: TLSOperational Resumed=false
52346mS HTTP: Secure Rx Src: 192.168.42.31(4096)-(443)
  GET /tpkt/openapi HTTP/1.1
  Connection: Upgrade

  Authorization: Basic ***** ← “TestApplication:password” encoded as Base64
  User-Agent: MyUserAgent 1.0
  Host: 192.168.42.11 ← Try to avoid populating “Host” header. It is un-necessary.
  Upgrade: websocket
  Sec-WebSocket-Key:
  Sec-WebSocket-Protocol: openapi
  Sec-WebSocket-Version: 13
HTTP: 192.168.42.31(4096)-(443) HTTPServerSessionIO: stCreationCallback(7)
HTTP: Public IP=192.168.42.31 Private IP=Not set
HTTP: 192.168.42.31(4096)-(443) HTTPServerSessionIO: stCreationCallback URI is authenticated
HTTP: ClientSessionsMgr::PopulatePwd(): Enter
HTTP: 192.168.42.31(4096) HTTPWebSocketUpgradeServerSessionIO: stCreationCallback
HTTP: 192.168.42.31(4096) HTTPWebSocketUpgradeServerSessionIO
HTTP: 192.168.42.31(4096) HTTPWebSocketUpgradeServerSessionIO: SetState Schedule
HTTP: 192.168.42.31(4096) HTTPWebSocketUpgradeServerSessionIO: SetState Proceed

52371mS CTI3: session=1 Start ← When successful, you see this
52371mS HTTP: Secure Tx Dest: 192.168.42.31(4096)-(443)
  HTTP/1.1 101 Switching Protocols
  Connection: Upgrade
  Server: IPOffice/WebSocketServer/
  Upgrade: websocket
  Sec-WebSocket-Accept: JU7m3Vkt8i15EzHqOXXGrxlnN5l=
  Sec-WebSocket-Protocol: openapi
  Sec-WebSocket-Version: 13

```

## First payload

This is a typical lines subscription

```

Message
{
  subscribe
  {
    requestid=1
    subscribe_id=1
    timeout=3600
    lines
    {
      flags=7
    }
  }
}

```

It should encode as exactly these 14 bytes:

```
1A 0C 08 01 10 01 18 90 1C C2 02 02 08 07
```

With framing, it should be:

```
00 00 00 01 1A 0C 08 01 10 01 18 90 1C C2 02 02 08 07
```

If you send it correctly, you will see it decoded on SysMonitor.

## Early releases of IP Office

Before Release 11.1.0.0, this interface is under controlled introduction, and may not be fully functional. Particularly, the error reporting does not really exist.

You need to add a NoUser source number for releases prior to 11.1.0.0

NoUser:

User

Voicemail

DND

ShortCodes

Source Numbers

Telephony

Forwarding

Dial In

Vo

Source Number

OPENAPI\_ALLOW

Additional features will be added with new releases.

In 11.1.0.0 there is a GeneralCmd called “getversioninfo”, which is a simple way to find out what release of IP Office you are connected to.

Something like this:

```

Message
{
  generalcmd
  {
    requestid=3000
    getversioninfo
  }
}

Message
{
  generaldata
  {
    responseid=3000
    versioninfo=IP Office 11.1.0.0 build 600
  }
}

```

It does not work on versions before 11.1.0.0

## Change History

Issue	Date	
1.0	15/4/2020	Initial Creation
1.1	01/08/2025	Updated for Release 12.2