Cisco Macro Scripting Tutorial

Doc: D1539401

The purpose of this tutorial is to show how to write your first macro scripts for CE video systems, and explain the examples in the editor in more detail.

No JavaScript knowledge is required, but you should have some experience with the XAPI and Cisco video systems. To write advanced macros or production quality macros, you may want to do a more in-depth JavaScript course, or seek advice from others.

For maximum learning effect, we recommend that you play with the macro editor while reading, and type in the examples manually and alter them to get into the rhythm.

XAPI Recap

A short recap of the features of the XAPI:

Status are properties in the video system that may change over time. You can get values on request, or get notified when they change. Examples: the current system volume, whether the system is in a call or whether someone is presenting.

Commands are the most common way to interact with the video system, such as making a call, starting a presentation or changing the mic volume. The commands usually lead to temporary changes, not kept after reboot. Typically a command causes a status value to change. Commands can accept parameters (eg who to call and the call rate).

Configurations are more permanent settings on the video system such as system name, default volume, proximity mode, wallpaper image etc. These settings remain when you reboot the system.

Event listeners let you listen to events, status values or configurations. Each time they change, you are notified with the new values.

Getting started

Before we start doing XAPI stuff, you should know how to debug. Write the following macro:

```
console.log('Hello Macro');
```

Save and run it. Notice that `Hello Macro` is written in your console log. This is useful during development to understand what is happening, what part of your code is being executed etc. Also, the logs are saved, so support teams can look later if something unexpected happened during usage.

Remember to end each JavaScript statement with a semicolon.

The xapi library object

The 'xapi' library lets the macros talk to the video system. To import it, simply type:

```
const xapi = require('xapi');
```

You are now able to able to use the `xapi` object in your code to invoke command, get status values, listen for events and edit configurations. You should only import it once per macro.

All the following examples require you to do this import first.

Invoke a command

To make a call with XAPI from TShell, you would typically do:

```
xCommand Dial Number: macro@polo.com
```

To do this from a macro:

```
xapi.command('Dial', { Number: 'macro@polo.com' });
```

Save and run. The macro should make a call as soon as it starts. Things to notice:

- The dot after `xapi` means you are calling a function from the `xapi` object, in this case `command`
- `Dial` is the first parameter to the function, and it is the xCommand that you want to

invoke

- The second parameter is the parameter to the command, in this case the number to call
- The argument is an object, therefore it is surrounded by `{}`
- `Number` is a property name for the argument object, and does not need apostrophes
- The command name and the property value are strings, and must have `''` around them

Error handling

It is good practise to always catch errors when using the `xapi` library. We skip this in most of the examples for readability.

```
function handleError(error) {
  console.log('Error', error);
}

xapi.command('Audio Incorrect Command').catch(handleError);
```

Usually in JavaScript, you need to read the code from the bottom and up. We define our functions and variables before we use them.

Multiple arguments

The XAPI command for swapping content on dual monitors with the XAPI is:

```
xCommand Video Matrix Swap OutputA: 1 OutputB: 2
```

To do this with a macro:

```
xapi.command('Video Matrix Swap', {
  OutputA: '1',
  OutputB: '2',
});
```

Notice that you separate the object properties with comma, and you can put them on separate lines for readability.

Events

In the previous examples, the commands were executed as soon as the macros started, which is usually when the video system has finished booting. That would probably not be very useful. It is more common that macros are listening to certain events, and perform actions when those events occur. An example is calling a number when an In-room Control button is pressed on a custom quick-dial panel.

To listen for an In-room Control button in XAPI we would do:

```
xFeedback Register Event/UserInterface/Extensions/Widget/Action
```

When you clicked a button, you would typically get an event like this:

```
*e UserInterface Extensions Widget Action WidgetId: 'quickdial'
*e UserInterface Extensions Widget Action Type: 'clicked'
*e UserInterface Extensions Widget Action Value: ''
```

To make a macro that dials a number when a specific widget is clicked:

```
function quickDial(event) {
  if (event.WidgetId === 'quickdial' && event.Type === 'clicked') {
    xapi.command('Dial', { Number: 'macro-polo@cisco.com' });
  }
}

xapi.event.on('UserInterface Extensions Widget Action', quickDial);
```

- First we define the function for setting up the call (called `quickDial` in this example)
- Then we connect the In-Room Control event and this function, using `xapi.event.on`
- The event handler will continue to listen as long as the macro is running
- The second parameter `quickDial` is the name of the function we want to call whenever an In-Room Control event occurs
- We check the action type and the widget id, so we don't start the call when other type of In-Room Control events occur
- Note the `===`, this means that the values must be exactly equal. JavaScript is weird.

Note that we send `quickDial` to tell the event listener the name of the function. This is known as a **callback** in JavaScript. If you typed `quickDial()` instead, this would not work as intended.

Stopping the feedback listener:

To stop your event listener:

```
const stop = xapi.event.on('UserInterface Extensions Widget Action', quickDial);

// ... Insert other code here

stop(); // stops listening to the event
```

`//` here means that the rest of the line are comments, the macros will not care about them but they are used to explain what your code does.

Status feedback

Events occur at singular points in time, such as the press of a button. You can also get notified on xStatus changes in a similar way to the events. To be notified when the system volume changes in the XAPI:

```
xFeedback Register Status/Audio/Volume/Level
```

A `const` keyword defines a constant. Once the macro starts, it cannot change. It is recommended to define them in the top of your macro, so they are quick to find and change. Whenever you type a value more than once, you should consider replacing it with a `const`.

To limit the value to 80% with a macro:

```
const MAX_VOLUME = 80;

function onVolumeChange(volume) {
  if (volume > MAX_VOLUME) {
    xapi.command('Audio Volume', { Level: MAX_VOLUME });
  }
}

xapi.status.on('Audio Volume Level', onVolumeChange);
```

If you need change a variable value during the lifetime of the macro, use `let` instead of `const`.

`const` is a hard-coded value, while `let` can change dynamically.

Status request

Sometimes you may need to ask for a status value directly, instead of listening to it constantly, with the `xapi.status.get` function.

To check if we are already in a call when the quick dial button above was pressed:

```
function callIfNotInCall(callCount) {
  if (callCount < 1) {
    xapi.command('Dial', { Number: 'macro@polo.com' });
    console.log('Call Macro Polo');
  }
}

function onInroomEvent(event) {
  console.log('In-room event occured', event);
  if (event.WidgetId === 'quickdial' && event.Type === 'clicked') {
    xapi.status.get('SystemUnit State NumberOfActiveCalls')
    .then(callIfNotInCall);
  }
}

xapi.event.on('UserInterface Extensions Widget Action', onInroomEvent);
```

The `then` function is a bit special. When we ask the video system for the number of calls, this request takes a little bit of time. Instead of blocking our processor, we return an object (called **promise**) that will call another function `callIfNotInCall` with the answer (number of calls) as soon as we get the response.

If you look at the log output, notice that the In-room event log comes before the call log, even though they appear in the opposite order in the code. This is because the program execution continues, and the result of the query is returned a few milliseconds later.

If this feels funky, just try to understand the syntax for it, since it is a common JavaScript pattern for doing responsive programming.

For more info, see:
Promise for dummies.

Alternatives way to write the code (syntax)

In the examples so far, we have given names to all our functions. You may also see functions without names in the examples in the editor, with an **arrow function** (`=>`) instead.

Eg:

```
function printVolume(volume) {
  console.log('Volume is', volume);
}

xapi.status.on('Audio Volume', printVolume);
```

could instead be written as:

```
xapi.status.on('Audio Volume', (volume) => console.log('Volume is', volume));
```

If you find this confusing, just stick to the more descriptive syntax. But it is nice to know the syntax if you see it in other code.

Advanced: Chaining your promises

Sometimes you need to do several other things before you decide to do an action, eg check whether you are in a call or there are people in the room (people count) before putting the system to sleep.

One way to do this is to 'chain' your promises like this:

```
function checkPeopleCount(people) {
  if (people < 1) xapi.command('Standby Activate');
}

function checkInCall(calls) {
  if (calls < 1) xapi.status.get('RoomAnalytics PeopleCount Current')
  .then(checkPeopleCount)
  .catch(console.error);
};

xapi.status.get('SystemUnit State NumberOfActiveCalls')
.then(checkInCall)
.catch(console.error);
```

You can also do this in a more compact, parallel way:

```
const p1 = xapi.status.get('RoomAnalytics PeopleCount Current', checkPeopleCount)
.catch(console.log);
const p2 = xapi.status.get('SystemUnit State NumberOfActiveCalls')
.catch(console.log);

Promise.all([p1, p2]).then(results => {
  const peopleCount = results[0];
  const callCount = results[1];
  if (peopleCount < 1 && callCount < 1) xapi.command('Standby Activate');
});
```

`Promise.all` continues when both of the results are available, and sends both results to `then` in an array. This is also slightly faster, since we don't wait for the first result before doing the second request. Again, this syntax may take some time getting used to. Go play with it :)

Configurations

Use `xapi.config.set` and `xapi.config.get` to set and get configurations. For example, to see if auto-answer is on:

```
xapi.config.get('Conference AutoAnswer Mode', v => console.log('Mode', v));
```

For configurations, you don't need to provide the parameter as an object, since there can be only one. To adjust the level of microphone 2:

```
xapi.config.set('Audio Input Microphone 2 Level', 33);
```

Config notifications are similar to status and event notifications. To print out when proximity mode is toggled:

```
xapi.config.on('Proximity Mode', v => console.log('Proximity changed to ', v));
```

Working with objects

Most of the previous examples use singular values, such as volume level, number of calls etc. You can also get objects back from `xapi`, and access the object's properties with a dot `.`:

```
xapi.status.get('Conference')
.then(conf => console.log(conf.DoNotDisturb, conf.Presentation.Mode));
```

If you want to see all properties of the object, just use `console.log` on it:

```
xapi.status.get('Video Input Connector', c => console.log(c));
// => { id: "2", Connected: "False", SignalState: "Unknown", Type: "DVI" }
```

Working with lists

The XAPI can also contain lists of elements. This is how you could find the input source of type 'PC' and start presenting it:

```
function present(connector) {
  xapi.command('Presentation Start', { ConnectorId: connector });
}

xapi.config.get('Video Input Connector')
.then(list => {
  for (let i = 0; i < list.length; i++) {
    const con = list[i];
    console.log(`Connector ${con.id}: ${con.Type}`);
    if (con.Type === 'PC') present(con.id);
  }
});
```

There are several things to note here:

1. This time, the result of our config query is a list, not a singular value
2. We can loop through the array with the `for` syntax you see above
3. The special ticks in `` `Connector ${i}` `` above lets us insert variables into the string
4. A Javascript array starts with index 0, but the XAPI list starts with 1, so we use the `.id` property of the list (instead of `i`) to get the correct connector

Timers

As you can see from the examples, reacting to events and updates is the preferred way to program macros. The alternative is to poll the system regularly, eg by asking it every ten

seconds whether it is in call or not. This is highly discouraged, as it is less responsive, more resource hungry and generally an error-prone solution.

Even so, there may still be good reasons to use timers, and we support the normal JavaScript methods for this, which is `setTimeout` to start a single timer and `setInterval` for recurring timers, and corresponding `clearTimeout` and `clearInterval` to stop them.

For example, to un-mute the system every hour:

```
function unmute() {
  xapi.command('Audio Microphones Unmute');
}

const everyHour = 60 * 60 * 1000; // In milliseconds
const timerId = setInterval(unmute, everyHour);

// If you want to stop the timer:
clearInterval(timerId);
```

Note again that we don't use parenthesis when referring to the callback `unmute` in `setInterval`.

Scheduling

There is no standard way in JavaScript, nor in the macro framework, to schedule actions at specific times of day. But you can copy this function into your macro and use it:

```
function schedule(time, action) {
  let [alarmH, alarmM] = time.split(':');
  let now = new Date();
  now = now.getHours() * 3600 + now.getMinutes() * 60 + now.getSeconds();
  let difference = parseInt(alarmH) * 3600 + parseInt(alarmM) * 60 - now;
  if (difference <= 0) difference += 24 * 3600;
  return setTimeout(action, difference * 1000);
}
```

Then, for example, to dial into standup every weekday at 09.00:

```
const StandupTime = '09:00';
const StandupUri = 'macro@polo.com';
const Sunday = 0, Saturday = 6;
```

```
function dialStandup() {
  const weekDay = new Date().getDay();
  if (weekDay !== Sunday && weekDay !== Saturday) {
    xapi.command('Dial', { Number: StandupUri });
  }

  schedule(StandupTime, dialStandup); // schedule it for the next day
}

schedule(StandupTime, StandupUri);
```

Notice that we schedule tomorrow's standup when calling in today, and then this repeats.

Performance and life cycle

The macro system has a built-in safety mechanism to prevent macros from causing performance problems. The macro **runtime** process is regularly measured. If the macros or the system in general is under heavy load, the runtime may be temporarily stopped, allowing the video system to continue to function optimally. Like 3rd party apps on a mobile phone, the OS might choose to stop aggressive or buggy macros without warning. After a short time, the runtime will automatically restart all enabled macros again.

Because of this, we recommend:

- Try to avoid having state in your macro, and detect it from the system instead. For example, instead of using a variable to remember if your system is in call or not, just ask the system for this when the macro starts.

- Don't do too much at the same time. Eg just when a system goes into call, a lot of resources are used. So if you have a lot of XAPI requests that you need to make and you see that the macro is getting killed, spread them out a little bit with `setTimeout`.

- Test the macro on the actual video system that you intend to run them on, and in similar or worse conditions. Test it in multi-site call with presentation etc to verify that everything runs smoothly.

User interface elements

Macros often need user interface elements, either to give users choices, or to give them information. Here are some of the most common components we provide, and examples on

how to use them. For complete reference, as always, see the full CE API guides.

Alert

An alert pops up with chosen text in the middle of both the video screen and the touch screen. The user can close it from the Touch, or it dismisses automatically after a timeout.

```
xapi.command('UserInterface Message Alert Display', {
  Title: 'Volume limiter',
  Text: 'A custom script is preventing you from setting the volume any higher',
  Duration: 10,
});
```

Notification

Notifications are very similar to alerts but less prominent, they only show on the main window screen, and in the top right corner of the screen.

```
xapi.command('UserInterface Message TextLine Display', {
  Text: 'Time left: 1 minute',
  Duration: 1,
});
```

Note that if another macro (or the video system) shows an alert or notification later, it will remove yours, so don't rely on it to show text permanently.

Prompt

The prompt pops up and lets the user select from a list of pre-defined choices. When the user makes a choice, this generates an event that you can react to. The `FeedbackId` lets you identify which prompt generated the answer (you could have more than one dialog in your macro). `event.OptionId` gives you the actual choice that the user selected.

Show a call quality survey after a call ends:

```
function showSurvey() {
  xapi.command('UserInterface Message Prompt Display', {
    Title: 'Call quality survey',
```

```
    Text: 'How was the quality of this call?',
    FeedbackId: 'call-quality',
    'Option.1':'Amazing',
    'Option.2':'OK',
    'Option.3': 'Terrible'
  });
}

xapi.event.on('UserInterface§ Message Prompt Response', (event) => {
  if (event.FeedbackId !== 'call-quality') return;
  console.log('Quality:', event.OptionId);
});

xapi.event.on('CallDisconnect', (event) => {
    if(event.Duration > 10) showSurvey();
});

showSurvey();
```

Currently, macros cannot store or send this information anywhere, but this may change in later releases.

Text input

Similar to the prompt, but lets the user type text instead of predefined choices.

This lets the user type in a name for the video endpoint:

```
xapi.event.on('UserInterface Message TextInput Response', (event) => {
  if (event.FeedbackId === 'system-name') {
    xapi.config.set('SystemUnit Name', event.Text);
  }
});

xapi.command('UserInterface Message TextInput Display', {
  FeedbackId: 'system-name',
  Title: 'Choose system name',
  Text: 'When you call someone, they will see this name',
});
```

In-Room Control

In-room Control provide a fully customisable user interface with widgets such as toggles, sliders, buttons, radio buttons, etc. It talks to the macros via the XAPI, just like the rest of the video system components.

All In-Room Control widgets have a unique **widget id**. When an action occurs, such as a button being pressed or a switch toggled, it generates an XAPI event that the macro can listen and react to.

Eg to adjust the volume with a slider instead of pressing the physical volume buttons:

```
function onUiAction(event) {
  if (event.WidgetId !== 'volume_slider') return;
  const newVolume = parseInt(event.Value * 100 / 255);
  xapi.command('Audio Volume Set', { Level: newVolume });
}

function syncUi(level) {
  xapi.command('UserInterface Extensions Widget SetValue', {
    WidgetId: 'volume_slider',
    Value: level * 255 / 100,
  });
}

// Listen to UI events:
xapi.event.on('UserInterface Extensions Widget Action', onUiAction);

// Sync the UI:
xapi.status.on('Audio Volume', syncUi);

// Correct initial value:
xapi.status.get('Audio Volume', syncUi);
```

Please see more in the macro editor's example section.

Note: In-room Control is intended to be a fully bi-directional API. This means that if you have made a macro / in-room panel for changing the ultrasound pairing, and someone changes the setting from somewhere else (another Touch 10, the web admin page, command line etc) you should make sure your ui is in sync. This means always confirming a status change (`Widget SetValue`), even if the originating GUI is showing the right state.

For a full reference guide to In-room Control, visit the In-room Control editor.

Tips

- Use events / feedback, not polling, to make your macros react.
- Macros can completely change the normal behaviour of the system. If your macro may surprise or confuse normal users, let them know with for example a notification on the

video system screen.

- We recommend that you don't create macros that depend on other macros. You can of course create several macros that listen to the same xapi values, eg the call state, as long as their actions are independent of each other. You have no guarantee for the order in which macros are executed.

- Currently, the macros cannot get or send data externally, eg to control lights in the room. For this you need an external control system. Even so, it can be useful to combine macros and external systems, such as a Crestron for low-level light control, and macros to adjust the light depending on presentation status, call status etc.

- If your macro gets uncomfortably big or complex, consider making it an external integration instead. The same code can be adjusted to run on a standalone Node server running JavaScript, for example.

What if I screw up?

The macro process will stop macros automatically if they run wild. You can then disable the macro while the macro framework is restarting. To completely disable macros, the kill switch is `xConfig Macros Mode: Off`.

What if I screw up badly?

If you make a macro that causes a boot when it starts up, it's a bit more tricky, since rebooting will cause the same problem as soon as the macro starts again.

To turn off macros while booting but before the macro process start, you can try this from the command line:

```
while sleep 1
do
  echo "xConfig Macros AutoStart: Off" | ssh root@your-ip "/bin/tsh"
done
```

Now, you can disable your bad macro or fix it, and restart the macro runtime. Remember to set autostart on again when the problem is solved.

Go on!

Make your own macros! Have fun! And please feel free to share them with us if you think they can be useful for more people.