Building Kerio Operator Provisioning Modules

for Kerio Operator 2.6.0

What Are Provisioning Modules?

A provisioning module is a self-contained package that handles provisioning for a set of similar phone models. Usually there is one module per phone manufacturer but there are exceptions: e.g. the Linksys and Cisco SPA phones are exactly the same (thus we put them into the same module to avoid code duplication) and both very different from the Cisco 79xx series (which has its own separate module).

Module Installation

There are two kinds of provisioning modules: *bundled* and *add-on*. Bundled modules come with Operator and take care of all the phones that we support "out of the box". Add-on modules can be installed by the user from special package files. Only modules signed by Kerio Technologies are allowed to be installed by default. This restriction is necessary as the module can do practically anything – e.g. set the phones' SIP proxy to an eavesdropping server instead of the Operator IP address. Bundled modules cannot be removed or updated (other than by upgrading Operator). However, they can be disabled and an add-on module can then handle the same phones in a different way. Inside, bundled modules look exactly the same as add-on ones. They have the same structure, use the same API and are not in any way "privileged" – bundled and add-on modules have the exactly same set of features available.

Modules can be installed, upgraded and removed in *Administration > Provisioned Phones > Advanced > Provisioning Modules*. To install unsigned modules (use only for testing your own modules!) hold down Shift while opening the *Provisioning Modules* dialog.

A Simple Provisioning Module

Throughout the documentation, we will be building a module that handles a small subset of the Cisco SPA phones. We will start with the very simplest version that can just configure the phone's lines and nothing more. Later, we will add additional useful features like dialplan, resync, support for the "Download Configuration" feature in Operator Administration, or support for phones with some typical problematic behaviour. You can find several versions of the example as it evolves attached to this documentation: both unpacked (directories `example-1/`, `example-2/`, ...) for convenient inspection and zipped (files `example-1.zip`, `example-2.zip`, ...) for easy installation directly to Operator.

Trying Out the Example

You can try out our example on any instance of Operator 2.2 or newer but you probably shouldn't use any production machines. First of all, be sure to *disable the original Cisco SPA module in the Administration*. Otherwise the bundled module might get used instead of the newly installed one. Then you can use the steps described in *Module installation* to install the example module. You can upload the zip file as you downloaded it and it will

just work. Or you can unpack it, make some simple changes, zip it again, upload the result and see your modifications come to life.

If you happen to have a Cisco SPA 3xx/5xx phone, you can test that the module actually provisions the phone. Otherwise, you can just look at the configuration files that the module generates. There are several ways of doing that; the easiest one is over HTTP.

That is only possilbe when your module responds also to HTTP requests. Our example does so but only for the purpose of being easy to try out. The SPA phones never use HTTP for provisioning and a real SPA module should respond only to TFTP requests.

To see the generated configuration file, open the URL

```
http://operator_ip_address/Cisco/SPA504G/112233445566.cfg
```

in your browser. You should see a text file beginning with:

```
<!-- This is the example SPA template, version 1. -->
```

If you see something else, you probably didn't disable the original SPA module (which currently listens on HTTP too). If you see a 404 error, you probably mistyped the path or didn't successfully install the example module.

After doing this, a new phone should appear on Operator's *Provisioned Phones* screen, with MAC address `11:22:33:44:55:66` and phone type *SPA example phone*.

Provisioning Module Structure

Each provisioning module is (until signed) a zip archive containing a nubmer of mostly plain-text files. The filenames inside the archive are **case sensitive** and we strongly recommend to make all of them lowercase unless there is a reason to do otherwise. The directory structure of our simple example looks like this:

```
+-- info
+-- phone_types
+-- providers/
|   `-- spa.php
`-- templates/
    `-- spa_main.tpl
```

It's often customary to put all the files in an archive into one toplevel directory, so a structure like that is also supported:

```
+--mymodule
   +-- info
   +-- phone_types
   +-- providers/
   |   `-- spa.php
   :
   .
```

(the toplevel directory can be named arbitrarily). This option was added in 2.2.0 RC1.

The `info` File

The `info` file is a simple INI file containg the most basic information about a module like its name and version. See the commented file in our example for an explanation of the individual options.

The `phone_types` File

The `phone_types` file lists all the phone types supported by your module. A phone type can represent one specific phone model (e.g. "Cisco SPA504G") or a whole group of similar phones (such phone types are usually titled *generic*). Each phone type declaration contains:

- A unique string identifier (a short lowercase name without spaces, e.g. "`spa504g`"), which will be used later as an argument to `associatePhone()`. It shouldn't change between module versions.
- Maximum number of lines allowed for phones with this types.
- A user-visible name (e.g. "Cisco SPA504G").
- Other optional parameters (e.g. resync type), which shall be discussed later.

One such record might look like:

```
spa3xx   3   ``Example SPA 3xx''   RESYNC=SIPNOTIFY
```

See the example for exact syntax.

The Provider(s)

Most modules will contain only one provider. It's possible to have more but let's forget about that for now. The provider should be named `providers/(something).php` where `(something)` is an arbitrary string, usually related to which phone types the provider handles (`spa` in our case). The extension is important. The provider is a PHP script and to create it, basic knowledge of PHP codiding and regular expressions is required.

The core "action" in provisioning is that phones ask for configuration files. We call each of these "askings" a *provisioning request*. Each request comes using one of the supported provisioning *protocols* (TFTP, HTTP or FTP) and asks for a configuration file with a specific *filename*.

Whenever Operator gets such a provisioning request, it runs the providers in all the installed provisioning modules in some order (which isn't important in most cases and we'll worry about it later). Each of the providers gets passed the request information (protocol, filename) and has several tasks:

- Determine whether the request is relevant (e.g. the SPA module should react only to filenames used by SPA phones).
  - If it is, generate the content of the configuration file.
  - If it is not, let other modules handle the request.

Provider Input

All information is passed to the module in form of global variables, which are set by Operator prior to invoking the script. It is very similar to how variables like `$_GET` work in web-oriented PHP. Most important variables are:

`$PATH` The filename of the requested configuration file including full path. Always starts with a slash, even if the phone doesn't originally include it. E.g. `'/Cisco/SPA504G/112233445566.cfg'`.

`$PROTO` The protocol using which the phone asked for the file. One of the strings `'tftp'` (most common), `'http'` and `'ftp'`.

`$OPERATOR_IP` The IP address of the Operator machine. You will probably want to set phone's *Outbound Proxy*, *Registrar* and similar options to this value.

There are also variables that convey some of Operator's global configuration parameters. Examples include `$TIMEZONE_OFFSET` (the current timezone's UTC offset in seconds) and `$VOICEMAIL_EXT` (the voicemail access extension, usually `50`).

The list is far from complete, more variables can be found in the reference guide and by examining Kerio's built-in provisioning modules.

Another bunch of important variables can be made available using `associatePhone()`, see below.

Provider Output

The exection of the provider usually ends by calling one of the *output functions*. Each of these functions returns some status to the provisioning system and/or sends some data to the phone; all of them terminate the script when done. When the end on the script is reached without calling any of these function, an implicit `passNext()` is assumed.

For a start, these two will be more than sufficient:

`passNext()` Call when the provider is not interested in the request. The request is passed on to the remaining providers.

`sendTemplate('template_name')` Processes the template `templates/template_name.tpl` to create a configuration file, which it then sends to the phone and ends processing. See the section on templates for more information.

There are several others (`sendStaticFile()`, `sendNotFound()`, `sendOk()`), which may come in handy later. See the reference guide for those.

associatePhone()

By calling `associatePhone('112233445566', 'sometype')`, you are telling Operator: "The currently requested configuration file belongs to a phone with MAC address `11:22:33:44:55:66` and phone type `sometype` (that is the identifier from the first column of `phone_types`).

This is used by Operator for two things:

- The phone is looked up by the supplied MAC address in the table of provisioned phones (i.e., the list you see on the *Provisioned Phones* screen in Administration).
- If no record is found, one is created and an extension is automatically generated for the phone (if enabled).
- The configuration for the phone (e.g. its lines) is loaded from the Operator databases and made available to the script in form of additional global variables.

The variables made available by `associatePhone()` include:

`$LINES` This variable contains an array of phone's configured lines. Each line has the form:

```
array(
     `TELNUM'         => '10`,
     'USER_FULLNAME'  => `Admin',
     'SIP_USERNAME'   => `10p1',
     `SIP_PASSWORD'   => 'eej5Shae1aeh',
     // ...
 )
```

`$ADMIN_PASSWORD` A password that should be used to protect phone's web administration if possible. Set in the provisioned phone properties, generated by default.

As always, the full list of available variables can be found in the reference guide.

You might wonder where you are supposed to get those magical *MAC address* and *phone type* values? The only information you have available is the *filename*. Decent phones (like the new SPA series) include both their type and MAC address in the configuration filename (e.g. `/Cisco/SPA504G/112233445566.cfg`). Most other phones can be forced to do so; more on that later. In case you were wondering: no, we can't just use the source MAC address of the packet as the phone may be (and often is) behind a router.

A Short Example

```
<?php
if (preg_match(`~^/Cisco/SPA3[^/]+/([0-9a-f]+)\.cfg$~i',
               $PATH, $matches)) {
    associatePhone($matches[1], 'spa3xx');
    sendTemplate(`spa_main');
}
```

This short snippet (together with the respective configuration template) is actually sufficient to successfully provision a phone. More complete usage can be found in the example module.

Templates

A template is a recipe for generating a text file, in this case, the phone configuration file. The template is fed phone configuration and other variables as input and is expected to give a ready-to-use configuration file as output.

Template Syntax

The template is a text file that allows you to mix PHP code and literal output in a more readable way than the `<?php ... ?>` markers. If you are familiar with templating engines used in web frameworks (e.g. Smarty), this is a very similar concept.

The template syntax is line-based and there are two kinds of lines:

- **Output lines** are all lines NOT starting with a '@' and are used to output text. Their contents will get written to the output as-is, with the exception of *variable expansions*. A variable expansion consists of a variable name (including the `$` symbol) enclosed in curly braces, and will be replaced with the value of the given variable in the output. E.g. if you write

  ```
  voicemail_ext = {$VOICEMAIL_EXT}
  ```

  in your template, you will get

  ```
  voicemail_ext = 50
  ```

  in the output. The contents of the braces doesn't have to be a simple variable name but can be any PHP expression. E.g. `{strtoupper($PHONE_LABEL)}` is a valid variable expansion.

- **'At' lines** start with a '@' and contain literal PHP code, which is executed as-is. They are most commonly used to put conditions and loops around output lines (i.e., to output something conditionally and/or repeatedly). For example, you could write:

  ```
  @ foreach ($LINES as $line) {
  We have a line with number {$line['TELNUM']}.
  @ }
  ```

  For a phone with two lines, 10 and 11, this will be output:

  ```
  We have a line with number 10.
  We have a line with number 11.
  ```

Template Input

The template has access to the same global variables as the provider script: both the ones set by Operator (`$VOICEMAIL_EXT`, `$LINES`, ...) and ones set by the by the provider (e.g. `$timezoneString` in our example module).

Simple Resync (SIP NOTIFY)

Usually phones load their configuration once after booting. *Resync* is a general term for all mechanisms that allow telling the phone: "your configuration has changed, please reload it". Without it, you would have to manually reboot the phone after each configuration change (e.g. adding a line).

The exact mechanism to accomplish this may vary among phones, wherefore you may configure a *resync method* for each phone type using the `RESYNC=some_type` option in `phone_types` (as shown in your example).

There is a single bundled resync method that comes with Operator: `SIPNOTIFY`. It uses `SIP NOTIFY` packets with the `Event: check-sync` header to resync the phone. An example of what these packets look like may be found in the SNOM documentation. However, this resync method is not restricted to SNOMs; 3/4 of the phone models you can meet understand these packets, including our SPAs. If your phone is one of them, getting resync to work is as simple as appending `RESYNC=SIPNOTIFY` to the corresponding lines in `phone_types`.

Some phones need to configure several additional options for the SIP notify mechanism (again in `phone_types`, see our example). Some of them are:

`SIPNOTIFY_auth` Set if the phone requires authentication for the resync requests. Accepted values: `'none'` (default), `'basic'`, `'digest'` and `'auto'`. Some phones (e.g. SNOMs, SPAs) even have an option in their configuration file governing whether to require authenticated resyncs or not.

`SIPNOTIFY_event` The event name, defaults to `check-sync`.

`SIPNOTIFY_arg` Additional arguments to the event, e.g. SNOMs require setting `SIPNOTIFY_arg="reboot=true"`.

If your phone doesn't understand SIP NOTIFY, it is possible to implement arbitrary custom resync methods that e.g. use the phone's web administration to force a resync or reboot. More on that later.

Operator invokes the specified method automatically whenever some configuration concerning the phone changes (a line is added to the phone, the user owning one of the phone's lines changes fullname, ...) or it can be run manually using the *Reload configuration* item in the context menu of the *Provisioned Phones* table in Administration.

Advanced Topics

Dialplans

Usually with IP phones, you have to end dialling a number by pressing some kind of "OK" button to tell the phone that there are no more digits to come and it can begin the call. This may be fine when dialling a long number that you will want to re-check twice before making the call. However, when calling a three-digit internal number, pressing an additional button might get annoying. That's where dialplans come in.

A *dialplan* is a string that tells your phone how long various extensions in your PBX are, thus allowing it to start dialling immediately after typing enough digits. Each phone has a completely different dialplan format and none of them is "major" like SIP NOTIFY is for resyncs. Therefore Operator doesn't generate any dialplans itself; it only gives you the data necessary to do it inside your module.

For each kind of dialplan that your script wishes to generate, create a script named `dialplans/(somename).php`.

Dialplan Script Input

The dialplan script receives a list of all the existing extensions and prefixes in the PBX as input, in the `$DIALPLAN_DATA` variable.

This variable contains an array with one entry for every extension or prefix. Each entry has the form:

```
array(`prefix', max_num_of_digits_that_may_follow)
```

E.g. for an installation with extensions `10`, `11` (plus the voicemail access number `50`) and an outbound route with prefix `0`, `$DIALPLAN_DATA` will be:

```
array(
    array(`10', 0),
    array(`11', 0),
    array(`50', 0),
    array(`0', -1),
)
```

Currently `max_num_of_digits_that_may_follow` is always `0` (for extensions) or `-1` (meaning "infinity"; for prefixes). However, this may change in future versions (e.g. we might automatically group the first two records into `array('1', 1)` or might support outbound routes with a fixed number length), so you should be prepared for any value.

Dialplan Script Output

The dialplan script should write the textual dialplan as understood by the phones to its standard output (e.g. with `echo` or `printf` calls).

Associating Dialplans with Phones

Each of the generated dialplans is available to the providers in the `DIALPLAN_(name)` variable (e.g. the dialplan generated by `dialplans/spa.php` is be available as `DIALPLAN_spa`). However, using these variables is not recommended unless necessary.

A better way is to set a dialplan type for each phone model in `phone_types`. To do so, just append `DIALPLAN_TYPE=spa` to the corresponding lines in `phone_types`. Then (after calling `associatePhone`, of course), the correct dialplan for the current phone will be available in the `$DIALPLAN` variable.

The second method has the advantage that when a dialplan changes (e.g. after adding a new extension), Operator knows which phones are affected and can notify the user about this fact.

Simplifying Dialplans

The ideal dialplan would simply contain all the existing extensions and prefixes, so the phone would always exactly know whether something can follow the already typed digits or not. However, the real world is a little more complicated: most phones (severely) *limit the length* of supported dialplan strings. A longer one either gets ignored completely or worse only a part of it is interpreted, making some numbers undiallable.

In the spirit of "half a loaf is better than no bread", Operator offers facilities to shorten the dialplan at the expense of accuracy.

Currently only one, to be precise. As there is usually no information available about the limits imposed by different phones on dialplan length/complexity and those limits may also vary between firmware versions, we had to make a wild guess on how much to simplify the dialplan. The result is `dialplanSimplifyOnedigit`.

`dialplanSimplifyOnedigit` takes the raw dialplan data as input and returns a simplified version (with the same structure) that has only "one-digit precision", i.e. the number of digits that can be typed until the phone automatically starts dialling will depend only on the first digit of the dialled number. For example, when you have extensions 10, 150, 20, 25, 99 and an outbound route 9, `dialplanSimplifyOnedigit` will perform the following transformation:

```
Original data             |       Simplified data
-------------             |       ---------------
                          |
array(                    |       array(
    array(`10',  0),       |           array(`1',  2),
    array(`150', 0),       |
    array(`20', 0),    ======>        array(`2',  1),
    array(`25', 0),       |
    array(`99',  0),       |           array(`9', -1),
    array(`9',  -1),       |       )
)                         |
-------------------------------------------------
               Example dialplan (regex)
-------------------------------------------------
10|100|20|25|99|9.*       |       1..|2.|9.*
```

Now when calling 10 or 99 you will have to end dialling manually but all the other extensions will be dialled automatically. With most usual numbering schemes, this simplification should give fairly good results. And as it always contains at most 12 items (one for each starting digit, `*` and `#`), it should be "short enough" for any phone.

If you are unsure what your phone can handle, we recommend using `dialplanSimplifyOnedigit`.

An Example

An example dialplan code for a hypothetical phone that uses ordinary regular expressions (as usually, of limited length) as its dialplan language:

```
$data = dialplanSimplifyOnedigit($DIALPLAN_DATA);
$first = true;
foreach ($data as $item) {
    if ($first) $first = false;
    else        echo ``|'';

    list($prefix, $maxLen) = $item;
    switch ($maxLen) {
        case 0:   echo $prefix;                         break;
        case -1:  echo $prefix . `.*';                  break;
        default:  echo $prefix . `.{' . $maxLen . `}'; break;
    }
}
```

A real-life example for the Cisco SPAs can be found in the second versions of our example module (`example-2`), look in `dialplans/spa.php`.

Handling Misbehaved Phones

Not all phones include both their exact model and MAC address in the filename they ask for by default. Some filenames are even used by several different phone types, e.g. both Polycoms and Yealinks ask for, among other files, `/(mac_address).cfg`. These situations can be usually resolved using the techniques below (and others). At least we haven't yet

found a phone that cannot be forced to send its model and MAC address, in a more or less kludgy way. But it's certainly possible such phones exist somewhere in the wild; if they do, they are probably unusable with Operator's provisioning (or pretty much any auto-provisioning system).

Two-Layer Configuration

Let's start with the easiest case. Some phones first ask for a "meta-configuration file", which describes the location of the main configuration file and all the others. Often you can use placeholders in the configured paths that will be replaced with the phone model and/or MAC addresss. A typical example are Polycoms: they first load a `(mac).cfg` with the following format:

```
<APPLICATION APP_FILE_PATH=``sip.ld''
    CONFIG_FILES=``polycom-[PHONE_MODEL]-[PHONE_MAC_ADDRESS].cfg''
    (...)>
</APPLICATION>
```

The `[PHONE_MODEL]` and `[PHONE_MAC_ADDRESS]` will be replaced by the respective values. This is a feature of the phone, not Operator. Immediately after reading this, the phone will ask for e.g. `/polycom-spip550-112233445566.cfg` and we are safe.

Implementing this in a provider will be as simple as:

```
if (preg_match(`~^/([0-9A-F]{12}).cfg$~i', $PATH, $matches)) {
    sendTemplate('polycom_meta');
} elsif (preg_match(`~^/polycom-([^/]+)-([0-9a-f]+)\.cfg~i'
                    $PATH, $matches)) {
    associatePhone($matches[2], $matches[1]);
    sendTemplate('polycom_main');
}
```

where `polycom_meta.tpl` contains the meta-configuration snippet shown above (which is just a piece of static text that doesn't use any templating features and will be sent to the phone as-is).

Redirection

Other phones don't have a special kind of configuration file to tell them where their configuration is; however, they allow to change the configuration file location from within itself. This is particularly the case of older SPA models, e.g. those made by Linksys.

By default, they ask e.g. for `/spa941.cfg`. When asked for this file, we can't send any actual settings (e.g. lines) because we don't know which phone is making the request. Instead, we output a small configuration file that just changes the configuration path (stored in the `Profile_Rule` option):

```
<flat-profile>
    <Profile_Rule>/Cisco/$PN/$MA.cfg</Profile_Rule>
    <Resync_Periodic>5</Resync_Periodic>
</flat-profile>
```

`$PN` and `$MA` will again be replaced with product name and MAC address by the phone. There is one more hack: we need to force the phone to load the configuration file *immediately*. To accomplish that, we configure it to reload its configuration every five seconds. We will revert this back to a normal value in the real configuration.

One advantage is that this "hack" is done only once, from then on, the phone remembers the new path and loads it directly. The disadvantage is that if the phone was previously used in a different provisioning environment, there will be some other value "stuck" in `Profile_Rule`

that Operator won't recognize (or vice versa). The only solution then is to change the value manually or factory reset the phone.

Implementation is very similar to the previous scenario and won't be shown here; additionaly, you can see it live in the third version of our example module (`example-3`).

Two-Stage Detection, Markers

Now comes the worst: phones that use weird filenames and don't allow changing them. In this case we assume that the phone sends at least its MAC address in at least one request (otherwise it would be unprovisionable) but doesn't allow easily detecting its type. Typical examples are phones that use the `/(mac_address).cfg` file. The type information, if discoverable at all, must be inferred from other files the phone ask for.

A typical example is presented by the Yealink/Well phones. They first ask for `/y0000000000(model).cfg` and then `/(mac).cfg`. The information we need is spread across two requests. Thus a mechanism for passing information between subsequent requests from the same phone is necessary. In Operator, *markers* are such a mechanism.

Markers are variables that are bound to a phone IP address for a short amount of time (currently 60s). When you call `setMarker('mymarker', 'value')` from within one request and another request comes from the same phone (IP address) within the time limit, the provider will get the `$mymarker` variable preset to `'value'` (in a similar way it gets variables like `$PATH`) and can react accordingly.

This warrants an example:

```
//             This file is requested first
if (preg_match(`~^/y0{10}([0-9]{2})\.cfg$~i',
               $PATH, $matches)) {
    setMarker('yealink_model', $matches[1]);

    // Simulate that the file doesn't exist
    // Phone will try the next one, (mac).cfg
    sendNotFound();
} elsif (isset($yealink_model)
         && preg_match(`~^/([0-9A-F]{12}).cfg$~i',
                       $PATH, $matches)) {
    associatePhone($matches[1],
                   'yealink_t' . $yealink_model);
    sendTemplate(`yealink_main');
}
```

Filename Conflicts, Module Sequence

As stated above, in most cases the order in which the modules are queried when a request comes is not important. Let's look at the situations where it is: when the same filename is used by several different phone type. A textbook example is our favourite `/(mac).cfg`. This filename seems to be very attractive for phone manufacturers – Polycom and Yealink we know of but there may be more. Let's look at the files each of them loads (let `11:22:33:44:55:66` be their MAC address):

Yealink T-32	**Polycom IP350**

`/y000000000032.cfg`
`/112233445566.cfg`
	`/112233445566.cfg`
	`/polycom-spip350-112233445566.cfg`
	`...`

Now if the Polycom module was run before Yealink, it would "steal" all `/112233445566.cfg`

requests, even those belonging to Yealink phones. The Polycom module doesn't know that some Yealink phones exist and Polycoms send nothing before the `/112233445566.cfg` based on which they could be distinguished. Therefore, the Polycom module has no choice but to respond to *all* `/(mac).cfg` requests; they look all the same to it.

The Yealink module on the other hand *can* recognize its requests from others: they are preceded by the `/y0000000000(model).cfg` request. In other words, Yealink's handling of `/(mac).cfg` is *more specific* or *selective* than Polycom's.

For this reason, we must ensure the Yealink module is run *before* the Polycom module.

Sequence numbers are a mechanism that allows that. Each module has a sequence number assigned. When a provisioning request comes, modules are sorted **lexicographically** (i.e., 15 comes before 2; this is done so that a new module can always be inserted between two existing ones without renumbering) by their sequence numbers and are run in that order. By default, all modules have a sequence number of `5` and there is no need to change it.

When there is such need, it may be set using the `sequence` option in the `info` file. E.g. Kerio's Yealink module's `info` contains:

```
sequence = 4
```

Do not change `sequence` unless you know what you are doing!

The *Download Configuration* Feature

There is an item in the context menu of the *Provisioned phones* table (right next to "Reload configuration") titled *Download configuration*. It allows the administrator to download a zip archive of all the configuration files used to provision a phone in order to manually inspect them, e.g. for couriosity and/or troubleshooting purposes.

In order to create such an archive, Operator needs to know which configuration files belong to a given phone. The providers can associate files to phones, now we need a mapping in the opposite direction.

That is set up using the `REVIEW` option in `phone_types`, which contains a whitespace-separated list of configuration file names that phones of the given type request, in the order they request them. Each filename has the format:

```
[?][!]proto:/some/path
```

Filenames prefixed with `?` are optional: if they dont't exist, they are silently ignored. Filenames prefixed with `!` are requested but are not expected to exist. All other files are required and an error is raised if they don't exist. `{$MAC}` will be replaced with the phone MAC address in the filename.

Let's use Yealinks, as described in Two-stage Detection, Markers as an example. The corresponding lines in `phone_types` might look like:

```
yealink_t18        1    ``Yealink SIP-T18P''
       REVIEW=``!tftp:/y000000000018.cfg tftp:/{$MAC}.cfg''
yealink_t20        1    ``Yealink SIP-T20P''
       REVIEW=``!tftp:/y000000000020.cfg tftp:/{$MAC}.cfg''
```

The `REVIEW` should be on the same line as the rest of the phone type definition; it's wrapped here only to fit on the page.

Internally, Operator sequentially simulates provisioning requests for the paths named in `REVIEW` from some random made-up virtual IP address, i.e. it emulates what the phone would do when loading its configuration. Whatever the providers return is put in the resulting configuration archive. This way, things like markers will work as expected and as long as `REVIEW` is set correctly, we can be sure that the files in the archive are exactly the same as what the phone gets: they are generated using the same mechanisms.

Efficiently Handling Larger Number of Phone Types

So far we have worked only with generic phone types in our example in order to keep our `phone_types` file short. However, it's usually better to create specific phone types (i.e., one phone type per phone model) when possible:

- The user will see the correct type in Administration.
- The correct maximum number of lines will be enforced.
- It's often needed to differentiate the phones based on support of some advanced features (e.g. SRTP, color logos, ...).
- It may be useful when implementing firmware upgrade (discussed later).
- It is considered generally "cleaner".

When using specific phone types, it is usually advised to include a generic "unknown" type in addition to the individual types. This way, when a new phone comes out after you released your module, it will work immediately, even though it would show as "unknown" in the user interface. Most of Kerio's built-in modules work this way.

In this section we shall discuss some features that will make maintaining a large number of phone types much easier. Use of none of them is in any way necessary: they were created chiefly to make your life easier. If you find any of them too complex or confusing, just don't use it.

The `$PHONE_TYPES` array

Most well-behaved phones send some kind of model name in their config, as seen on the SPA example. We can easily parse the model name from the filename using regular expressions. Now when a request for e.g. `/Cisco/SPA504G/112233445566.cfg` arrives, it would be good to do something like: "if a phone type named `spa504g` exists, use it for `associatePhone()`, otherwise use `spa_generic`". This would give us the forward compatibility of supporting arbitrary not yet known phones (as long as they keep the filename format and understand our configuration files) as generic while still using the "right" types for phones that we know.

And we want to do that without re-enumerating all the supported phones again in the provider script.

To do that, the `$PHONE_TYPES` variable is available to the providers. It's an array of all the phone types supported by the current module (as defined in `phone_types`). It's keys are the type identifiers (first column from phone types) and values contain additional information about the types (maximum number of lines, the `name=value` options set in `phone_types` for the given type, etc.).

The only thing you will be using this array for most of the time is testing a phone type's existence with `isset($PHONE_TYPES[$sometype])`.

Now we have all we need to create a simple specific/generic provider for the SPAs:

```
if (preg_match(`~^/Cisco/(SPA[^/]+)/([0-9a-f]+)\\.cfg$~i',
               $PATH, $matches)) {
    $model = strtolower($matches[1]); $mac = $matches[2];
    if (isset($PHONE_TYPES[$model])) {
        associatePhone($mac, $model);
    } else {
        // We don't know this phone - use the generic type
        associatePhone($mac, 'spa_generic');
    }
}
```

`@set` and `@unset`

Sooner or later you will get bored of typing:

```
spa501g 8 ``Cisco SPA501G'' RESYNC=SIPNOTIFY SIPNOTIFY_auth=digest
spa502g 1 ``Cisco SPA502G'' RESYNC=SIPNOTIFY SIPNOTIFY_auth=digest
spa504g 4 ``Cisco SPA504G'' RESYNC=SIPNOTIFY SIPNOTIFY_auth=digest
# ...
```

This is where `@set` and `@unset` come in.

`@set` and `@unset` are directives that can be used in `phone_types` and the use of `'@'` **bears no relation** to the `'@'` used in templates. `'@'` is used simply because it's easy to spot visually.

`@set` sets some options for all the following phone types. Each option persists until changed with another `@set` or unset with `@unset`. Options specified on the individual phone type lines override those set with `@set`.

Now the code above becomes:

```
@set RESYNC=SIPNOTIFY SIPNOTIFY_auth=digest

spa501g 8 ``Cisco SPA501G''
spa502g 1 ``Cisco SPA502G''
spa504g 4 ``Cisco SPA504G''
# ...

@unset RESYNC SIPNOTIFY_auth
```

Any `@unset`-s at the very end of `phone_types` may be omitted. Wildcards are allowed with `@unset`: you can write e.g. "`@unset RESYNC SIPNOTIFY_*`" or even "`@unset *`" to cancel any `@set`-s still in effect.

Variable expansion in `phone_types`

You may have noticed that `@set` as introduced above cannot be used e.g. for setting `REVIEW`, which is never exactly the same for each phone type as it must contain some indication of the phone model. However its values are usually very similar and we don't want to type them again and again. Let's take the Yealinks as described in Two-Stage Detection, Markers as an example:

```
yealink_t28 6 ``Yealink SIP-T28P''
                    REVIEW=``!tftp:/y000000000000.cfg tftp:/{$MAC}.cfg''
yealink_t32 3 ``Yealink SIP-T32G''
                    REVIEW=``!tftp:/y000000000032.cfg tftp:/{$MAC}.cfg''
yealink_t38 6 ``Yealink SIP-T38G''
                    REVIEW=``!tftp:/y000000000038.cfg tftp:/{$MAC}.cfg''
# ...
```

The values differ from one another by *two characters*. And believe us, most real `REVIEW` strings are *much* longer. We need a way to include "placeholders" in the common `@set` value that will be replaced with specific values for each individual phone type (in this case the model number).

These placeholders are called *variable expansions*. Variable expansion has exactly the same syntax as in templates: a variable name (or PHP expression) in curly braces, e.g. `{$somevar}`. As you have probably guessed, we have already see an example of variable expansion: the `{$MAC}` used in `REVIEW`. Now our example becomes:

```
@set REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''
yealink_t28 6 ``Yealink SIP-T28P'' model=00
yealink_t32 3 ``Yealink SIP-T32G'' model=32
yealink_t38 6 ``Yealink SIP-T38G'' model=38
# ...
```

As you can see, the variable expansions in `REVIEW` can reference values set for the individual types, even though they are defined "lower" in the file. This is called *late expansion*. It works because variable expansion is done separately for each phone type *after* the `@set` mechanism. In this example, the `@set` transforms the lines above into:

```
yealink_t28 6 ``Yealink SIP-T28P'' model=00
              REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''
yealink_t32 3 ``Yealink SIP-T32G'' model=32
              REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''
yealink_t38 6 ``Yealink SIP-T38G'' model=38
              REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''
# ...
```

It just adds the variable to all the phone types, without knowing anything about expansion. Later, when the value is needed, it is expanded. Then e.g.

```
yealink_t38 6 ``Yealink SIP-T38G'' model=38
              REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''
```

becomes

```
yealink_t38 6 ``Yealink SIP-T38G'' model=38
              REVIEW=``!tftp:/y000000000038.cfg tftp:/112233445566.cfg''
```

as one would expect.

Variable expansion can reference:

- Any options set in `phone_types`, either for one specific phone type or with `@set`. These can be:
  - Real Operator options (`RESYNC`, `REVIEW`, etc.). You probably won't find much use for this.
  - **Custom options.** Any `name=value` pair in `phone_types` that is not a known Operator option is a custom option but it's recommended to name them lowercase to avoid clashes with future Operator options. That's `model` in our example. This is probably the most useful thing to use in variable expansion.

    The custom options are available not only for `phone_types` variable expansion but also as variables in your provider and resync scripts!
- Implicit variables provided by Operator. Only one seems to be of any interest:
  - `$IDENT` – contains the identifier (first column) of the relevant phone type.

With the use of `$IDENT`, we could further shorten our example. We see that the model number information is duplicated: once in the identifier and once in `model=`. We could write this instead:

```
@set model=``{substr($IDENT, 9)}''
@set REVIEW=``!tftp:/y0000000000{$model}.cfg tftp:/{$MAC}.cfg''

# Exception: T-28 uses ``00'' instead of ``28'' in the filename.
yealink_t28 6 ``Yealink SIP-T28P'' model=00
yealink_t32 3 ``Yealink SIP-T32G''
yealink_t38 6 ``Yealink SIP-T38G''
# ...
```

Here you can see an application of the rule "options set for a specific type override those set with `@set`". Almost all Yealinks use the model number from their name in the filename; only historical T-28 uses all zeroes. Therefore we set `model` explicitly for T-28; all the others will inherit `model="{substr($IDENT, 9)}"` from the `@set` and will set their model number from their identifier.

This may almost seem unnecessary but when you have twenty different models with names a little longer than six characters, you'll appreciate it. And with SPAs, it can be used quite directly:

```
@set REVIEW=``tftp:/Cisco/{strtoupper($IDENT)}/{$MAC}''
```

See the third version of our example module (`example-3`) for a real-life example of the specific/generic phone type model, variable expansion in `phone_types`, `REVIEW` and misbehaved phone redirection.

Usage of expansion isn't limited to `REVIEW` and `@set`: it can be used in any of the `phone_types` options.

Implementing Custom Resync Mechanisms

The `RESYNC` option in `phone_types` may be set either to a built-in resync method (customarily named uppercase, currently only `SIPNOTIFY`) or the name of a resync script (preferrably lowercase). E.g. when you set `RESYNC=spa`, Operator uses a script `resync/spa.php` to perform the resync.

Resync Script Input

When started, the resync script has access to the full phone and Operator configuration, i.e. almost the same variables as what the provider has *after* calling `associatePhone()`: `$LINES`, `$ADMIN_PASSWORD`, `$VOICEMAIL_EXT`, ... Request-related variables (like `$PATH`) are not available as they make no sense in this context. Additional variables that might be of interest:

`$LAST_CONFIG` The most useful variable of all. Contains the last configuration successfully read by the phone. Keys are configuration variable names, i.e. you have `$LAST_CONFIG['LINES']`, `$LAST_CONFIG['ADMIN_PASSWORD']` and so on. This is important e.g. when the user is changing the phone administration password. `$ADMIN_PASSWORD` will contain the new password but the phone doesn't know about it yet, so it will ask for the old one. Therefore, you have to use `$LAST_CONFIG['ADMIN_PASSWORD']` when authenticating your resync.

`$MAC`, `$IDENT` The phone MAC address and type (i.e., whatever was passed to `associatePhone()` when the phone was created).

You will probably find these values most useful:

- `$LAST_CONFIG['ADMIN_PASSWORD']`
- `$LAST_CONFIG['LINES'][0]['SIP_USERNAME']`
- `$LAST_CONFIG['LINES'][0]['SIP_PASSWORD']`

HTTP(S) Resync

The most common resync method after SIP NOTIFY is using some kind of "reload configuration" option the phone's web administration, or, if that isn't available, most phones have at least a "reboot" button there. It's just a matter of sending the right HTTP requests. Fortunately, almost all phones currently use HTTP authentication to secure their administration, which is much easier than emulating .

The function allowing this is:

```
urlopenPhone(`scheme', '/path?query',
             array(`additional' => 'options'));
```

Currently only `'http'` and `'https'` schemes are allowed. You see that the IP address is not mentioned anywhere – it will be filled in automatically. For example, to request `http://<phone_ip>/admin/resync?`, call:

```
urlopenPhone(`http', '/admin/resync?`, array(...));
```

Additional options may contain the following:

`user`, `password` Authentication information.

`auth` The authentication type. `'auto'` (default), `'basic'` or `'digest'`. `user` and `password` are required when this option is set.

`body` The request body. When this option is set, a POST request is sent instead of the default GET.

`verifySslCert` Set to `false` to disable SSL certificate verification. Almost always needed with HTTPS.

The function returns a file handle from which you can read the reply using standard PHP functions (`fread` and friends). You **have** to read the whole reply and close the handle, even if you are not interested in it.

An example HTTP resync for the SPA phones (which has the advantage that unlike SIP NOTIFY it works even when the phone has no lines assigned):

```
$fh = urlopenPhone(`http', '/admin/resync?`, array(
         'user' => `admin',
         'password' => $LAST_CONFIG[`ADMIN_PASSWORD']));
while (!feof($fh)) fread($fh, 8192);
fclose($fh);
```

A complete example of a resync script can be found in the fourth version of our example module (`example-4`).

A Completely Generic Resync

In the unlikely scenario that your phone has some completely alien resync mechanism that is neither SIP NOTIFY nor HTTP(S), we offer a generic mechanism. Call:

```
$sock = connectPhone(`proto', port);
```

and you will get a socket connected to the phone using the given protocol (`'tcp'` or `'udp'`) and port, which can be manipulated using the usual PHP socket functions (`socket_send`, `socket_recv`, ...) in an arbitrary way. E.g. to connect using SIP:

```
$sock = connectPhone(`udp', 5060);
```

The ultimate example is Kerio's implementation of the SIP NOTIFY mechanism, which is built using the exactly same API. You can find it in the

```
/opt/kerio/operator/lib/php/provisioning/api/util_resync_sipnotify.php
```

directory of your Operator installation. The notes at the end of the documentation describe how to access such files. Don't forget to close your sockets when done.

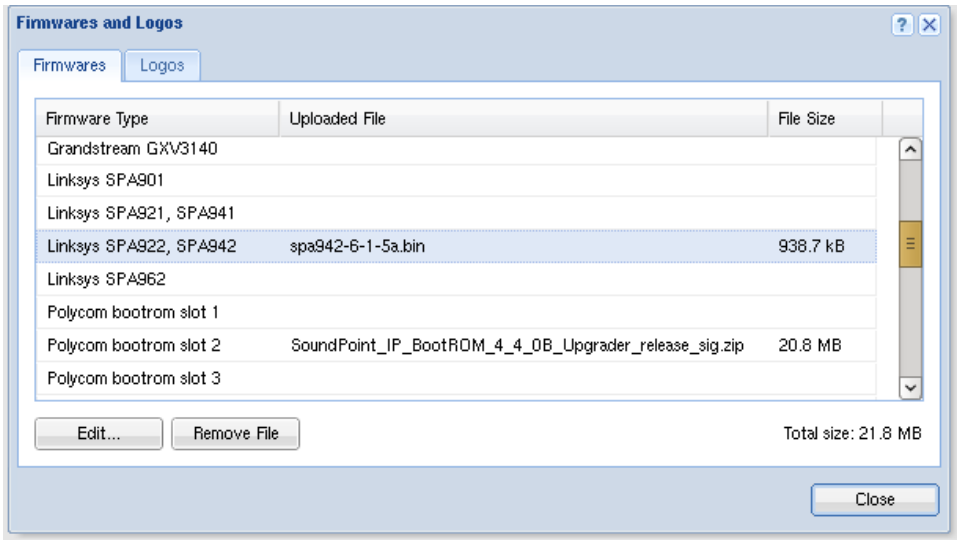
Firmwares and Logos

Some phones allow their firmware to be upgraded using provisioning. Some allow setting a logo to be shown on their screens (which you are probably aware of because of the Kerio Operator logo shown there by default). Surprisingly, these two tasks require doing very similar things on the server side, therefore they are implemented as one feature in the Operator provisioning system, called *Firmwares and Logos*. It allows users to upload such files, which are then sent to the phones during provisioning.

We will use logos as an example to introduce all the basic concepts. All of them apply to firmwares, too, however, with firmwares one has to be careful about a lot of other things that will be discussed later.

Slots

This is the *Firmwares and Logos* dialog from Operator Administration:



Each row in the table is a slot that can hold one uploaded file of a given kind (a firmware image for a given class of phones, a logo with a given resolution and color depth, etc.). When a new file is uploaded to the same slot, any previous content is erased.

Usually, there is one slot for each ”kind” of file supported. With logos, there usually one slot for every allowed combination of dimensions and color depth. Whatever you upload to this slot will be used on ALL phones with compatible logo format. It's not possible to set a different logo for one individual phone.

Defining Slots

Creating a new firmware/logo slot is very simple. It consist of adding a single line to the `firmware_types` file in your module. It has the same basic syntax as the familiar `phone_types`, only different columns. Consider an example:

```
@set SCRIPT=logo
logo_bw     ``Example SPA b/w logo''   logo               \
            LONGDESC=``Monochromatic 128x48 logo image''
logo_color  ``Example SPA color logo'' logo               \
            LONGDESC=``Color 320x240 logo image''
```

The values are, in order: unique identifier, human-readable name (shown in the firmwares dialog), type (*firmware*, *logo* or *other*) and additional *name=value* parameters (`LONGDESC` is an informative text shown in the UI to suggest users what kind of files to upload, for `SCRIPT` see below).

Verification & Preprocessing

Whenever the user uploads a file to a slot, it is not directly saved. It is first processed using a script, which has two main tasks:

- Verify that the uploaded file is correct. E.g. check if the user didn't upload a text file instead of a picture.
- Preprocess the uploaded file to a form in which it can be offered to the phones. E.g. convert and uploaded logo to the correct file format and resolution.

The verification and conversion is done only once upon uploading the file. Afterwards, the logo is stored inside Operator in the already converted format for efficiency reasons.

The Preprocessing Script

The preprocessing script can be either a PHP script or a Unix shell (i.e., bash) script. The latter is strongly recommended, as various file manipulation and invocation of external tools (e.g. for image processing) tends to be much easier in the shell.

The script should be saved as `firmware/<name>.sh` or `firmware/<name>.php`, where `<name>` is the value of the `SCRIPT` parameter from `firmware_types` if specified (this allows using one script for several slots), the slot identifier otherwise.

The script gets the following variables as input (global variables in PHP, environment variables in the shell):

- `$FW_INFILE` – the original uploaded file.
- `$FW_OUTDIR` – the directory to which to save preprocessed output.
- `$FW_VERIFY` – set to `0` when user wishes to relax file verification, `1` otherwise.
- `$FW_IDENT` – the identifier of the slot we are uploading to. Useful when one script handles multiple slots.

This script should perform some checks on `$FW_INFILE`. If it's deemed valid, it should be processed in any manner appropriate and the result should be saved to `$FW_OUTDIR` under whatever name(s) you choose. You can use the `$FW_OUTDIR/_tmp` directory for storing temporary and intermediate files. It is automatically created before your script is run and deleted after it ends, successfully or not. When finished, terminate your script by calling the `accept_firmware` function (`acceptFirmware()` in PHP). The whole `$FW_OUTDIR` directory is stored as-is by Operator and made available to provider scripts as described below.

Your script is run with `$FW_OUTDIR` as its working directory so you rarely have to reference this variable directly. Simply use relative paths.

If the file is not considered valid, `reject_firmware` (`rejectFirmware()`) should be called, also exiting your script. Upon rejecting a firmware, the whole of `$FW_OUTDIR` is automatically cleaned up so you don't have to worry about leaving half-processed files around.

A trivial script that does no checks, no preprocessing, and simply stores the uploaded file as-is might look like this:

```
cp $FW_INFILE logo.bmp
accept_firmware
```

Or in PHP:

```
<?php
copy($FW_INFILE, `logo.bmp');
acceptFirmware();
```

Serving the Files to Phones

There are two ways in which the phones access files such as firmwares and logos:

- For some phones, the path of the file is specified in the phone configuration file. This seems to be the more common case.
- Others load these files from hard-wired locations.

Fixed Filenames This is the easy case. For example, if the SPA phones always downloaded the logo from `/spa_logo_bw.bmp` (they don't), you could write something like this in the provider:

```
if ($PATH == `/spa_logo_bw.bmp' && haveFirmware('logo_bw')) {
        sendFile(getFirmwareFile(`logo_bw', 'logo.bmp'));
}
```

`haveFirmware()` checks whether there is a file uploaded in the given slot (specified by the identifier from the first column of `firmware_types`).

`getFirmwareFile()` returns the on-disk physical path of a file in a firmware slot. The first parameter is again the slot identifier, the second says which file from the preprocessed firmware (which is a directory) we want. `sendFile` sends the contents of the file to the phone (i.e., to the phone it will seem as if there were a file called `/spa_logo_bw.bmp` stored on the TFTP server, with the same contents as the `logo.bmp` created by the preprocessing script).

Arbitrary Filenames In fact, the SPAs (along with most phones) expect the logo filename to be specified in their configuration file:

```
<BMP_Picture_Download_URL>tftp://1.2.3.4/some/file.bmp</BMP_Picture_Download_URL>
<Select_Background_Picture>BMP Picture</Select_Background_Picture>
```

The filename can be completely arbitrary. However, it should contain:

- A prefix unique for your module, to prevent name clashes with others. We recommend using `/prov/$MODULE_ID/`.
- The slot identifier (e.g. `logo_bw`), so that we know what file is actually requested.
- A unique identifier that changes every time the user uploads a new file to the slot. This is necessary because many phones download each logo/firmware only once and then use the local copy until the filename changes. Such an identifier is automatically provided by Operator, see below.

The provider needs to do the following when generating the configuration file:

(1) Determine *which* logo slot should be used with the given phone.

(2) Check whether there is a logo uploaded in that slot.

(3) If it is, generate a filename for it and put it in the configuration file.

The association between phone types and logos/firmwares is done using the `FW` variable in `phone_types` (it may contain multiple comma-separated values, e.g. a logo and a firmware):

```
spa502g         1    ``Example SPA502G''    FW=logo_bw
spa504g         4    ``Example SPA504G''    FW=logo_bw
spa525g         4    ``Example SPA525G''    FW=logo_color
# ...
```

Then, you need to edit the provider:

```
// ...
associatePhone(...);

$firmwares = phoneFirmwares(`logo', true);

if ($firmwares) {
        $logoSlot = $firmwares[0];
        $logoId   = $FIRMWARE_TYPES[$logoSlot][`FW_CONTENT_ID'];
        $logoPath = ``/prov/$MODULE_ID/logo/$logoSlot/$logoId.png'';
}
```

The `phoneFirmwares` function accomplishes steps (1) and (2). It returns the list of firmware slots associated with the current phone type (it must be called after `associatePhone()` in order to know the phone type), further filtered as specified in the arguments. The first argument says that we are interested only in logos (not firmwares), the second that we want only non-empty slots (that have a file uploaded).

E.g. for a SPA502G phone, the return value will be `array('logo_bw')` if there is a black and white logo uploaded, `array()` otherwise.

`$FIRMWARE_TYPES['slot_identifier']` contains additional information about the slot and its contents (see the Reference Guide). One of its fields, `'FW_CONTENT_ID'`, is set to a random hexadecimal identifier each time a new file is uploaded to the slot and thus guaranteed to change with each upload.

And finally, you should edit the configuration template to use the computed `$logoPath` variable and put the right directives in the configuration file:

```
@ if ($DISPLAY_LOGO && $logoPath) {
     <BMP_Picture_Download_URL>tftp://{$OPERATOR_IP}{$logoPath}</BMP_Picture_Download_URL>
     <Select_Background_Picture>BMP Picture</Select_Background_Picture>
@ }
```

Operator allows the administrator to globally disable displaying logos on phones. This setting is reflected in the `$DISPLAY_LOGO` variable, which your module should respect.

Now all that remains is to actually send the logo to the phone when it asks for it. That's almost exactly the same as in the fixed filename case:

```
if (preg_match(``~^/prov/$MODULE_ID/logo/([^/]+)/([^/]+).bmp~i'',
                $PATH, $matches)) {
        $logoSlot = $matches[1];
        if (haveFirmware($logoSlot)) {
                sendFile(getFirmwareFile($logoSlot, `logo.bmp'));
        } else {
                sendNotFound();
        }
}
```

That's all! You can try the fifth version of our example, you should be able to upload a logo and have it shown on the phone's screen.

Preprocessing Logos

Most phones accept logos only in one exact combination of file format, resolution and colour depth. Usually an obscure one like a variant of 4-bit BMP that no ordinary user knows how to create. Instead of explaining this tedious process to everyone, you can allow uploading logos in arbitrary fomats and convert them to the right one in the preprocessing script.

For that reason, Operator contains the netpbm image manipulation toolkit. It is a collection of small standalone programs each of which performs a simple image operation. Most of them behave as filters, i.e. they read from standard input and write to standard output, allowing them to be easily chained into pipelines. You may find the following especially useful:

`<format>toppm` and `ppmto<format>` Convert between various image formats and PPM. Thus to convert e.g. from PNG to BMP, you can use `pngtoppm | ppmtobmp`.

`ppmquant` Reduce the color depth of images. Phones are very picky about color depth.

`ppmscale` Change the dimensions of an image.

And many more. See the netpbm manual for details. A few additional helper functions are provided by Operator:

`anytoppm <in-file> <out-file>` Autodetect the format of the input image (based on file header, not the extension) and convert it to PPM.

`ppmfix` Some of the `netpbm` tools output invalid PPMs when the image contains only two colors. Some other crash when given such as input. It may be needed to add `ppmfix` to various places in the pipeline (esp. after `ppmquant` and before `ppmto<format>` but possibly elsewhere, too). This may require some experimenting. When desperate, you can always put a `ppmfix` between every two `netpbm` commands. A good way of testing this is trying to upload an all-black image.

An example autoconverting preprocessing script for SPAs might look like:

```
anytoppm $FW_INFILE _tmp/logo.ppm || reject_firmware
case $FW_IDENT in
  logo_bw)
    # BMP 128x48 (1-bit)
    cat _tmp/logo.ppm | pnmscale -width 128 -height 48 \
                      | ppmquant 2 | ppmdist | ppmfix  \
                      | ppmtobmp >logo.bmp
  ;;
  logo_color)
    # BMP 320x240 (1, 4, 8, 24, or 32-bit)
    cat _tmp/logo.ppm | pnmscale -width 320 -height 240 \
                      | ppmfix  | ppmtobmp >logo.bmp
  ;;
esac

cat $TARGET | bmptopnm | ppmfix | pnmtopng >_preview.png
accept_firmware
```

The optional `_preview.png` file is used by the Administration to show the user a preview of the converted logo.

Firmwares: A Word of Warning

Firmware updates are handled in pretty much the same way. User uploads an upgrade image, it is verified and preprecessed and the resulting file(s) is offered to the phone under either fixed or configured filename(s).

There are some additional caveats though:

- Not all phones support it. And even for those that claim to support it, it doesn't always work.
- It is almost always poorly documented or undocumented. Usually a lot of experimenting and packet sniffing is required to understand the process.

- There are whole classes of phones that share a common firmware. E.g. Cisco releases only one firmware image for all SPA 30x and 50x phones (which are essentially the same phone, differing only in number of lines and the presence of an LCD). This usually corresponds to one firmware slot in Operator.
  - However, sometimes the classes are not clear-cut and even overlap and change with time. Polycoms are extremely notable for this. In one version, VVX 1500 shares firmware with SoundPoint phones, in another, with the other VVXs. As we can never be sure what the "groups" will be in the next version, we cannot create any fixed per-group slots in Operator. Instead, we offer three generic numbered slots to which you can upload any firmware images you like and using some magic with filenames the phones will work out what to download themselves. See Operator's Polycom module source code to get a better picture.
- The provisioning system itself doesn't know which phones need upgrading and which already have the new version. Thus, the upgrade is usually unconditionally offered to all the phones of the corresponding type, every time they ask. It's up to each phone to detect whether it should actually perform an upgrade, i.e., whether the offered firmware is different from its own. There are various ways of achieving that and all have some implications for how the provisioning module should behave.
  - Some phones start downloading the firmware, inspect the header for the version and if it's the same, immediately abort the transfer. This seems to be the most common variety.
  - Some phones only check whether the firmware *filename* specified in the configuration changed. If it's the same as last time, they don't even try to download the file. Thus it's important to give each uploaded firmware a unique virtual filename (this was already discussed in the section on logos, see above).
  - There may be even ones that always download the whole file before comparing anything. In such cases just having a firmware image uploaded might significantly slow down phone boot.
  - Perhaps there are even ones that always perform an upgrade, even if the target version is exactly the same. We haven't met any examples of that so far.
- Almost all phones do some sanity checking on the firmware to see that it's usable with the given phone, wasn't corrupted during transfer, etc. However, there is no guarantee of this, so be careful regarding what you are sending to the phone. In the extreme case, you could end up bricking your phone.

  Therefore we recommend to do at least some basic verification on the uploaded file. This involves a lot of guesswork as the firmware structure is practically *always* undocumented. A partial check may be better than nothing: e.g. if the firmware is distributed as a zip archive, you can check filenames inside the archive. You don't have to be afraid of being "too strict": the user can override the check in case of false positives. It serves more as a warning to someone who mistakenly uploads firmware for a different model or a `.exe` updater instead of a `.bin` image.
- Sometimes the steps needed to upgrade the firmware may depend on the firmware version *currently present* on the phone (which the provisioning system usually doesn't know). For example, it may not be possible to directly upgrade from firmware 1.1 to 2.0, instead having to go through 1.2. This can almost never by handled by the provisioning system and the administrator has to tak care of this himself; however, if you know such a problem, it's good to at least warn about it in the slot's `LONGDESC`.
- Some firwares are distributed as one opaque binary file. These are easy to understand and work with. Others have the form of a zip archive that has to be unpacked and the phone downloads each of the files individually. That is one reason why there is a whole directory for the preprocessed firmware and not just a single file. Tools like `unzip`, `tar`, `find` are at your disposal in the preprocessing script to unpack the archive, select the right files from it, etc.
  - This can often get tricky. Polycoms once again offer a *great* example. Their firmware zips contain everything: firmware images themselves, configuration files,

background pictures, etc. You must very carefully filter the archive contents to decide what files to offer to the phone. For example, you need to prevent the configuration files distributed with the firmware from being used instead of the generated ones. Whitelisting is usually better than blacklisting in these cases. Again, see our Polycom module for details.

The hard part is working out the logic. Implementing it is usually a few lines of code.

Changelog

2.5.3

- Fixed a PHP syntax error (extra curly brace in `providers/spa.php`) in examples 1 through 4 introduced in 2.3.1.
- Fixed the *Download Configuration* functionality in example modules.

2.3.1

- Added firmwares/logos support. Some support had been available since 2.2.0 beta 3 but only from 2.3.1 all the features listed here are available and documented.

2.2.2

- Changed the dialplan API to use `dialplans/*.php` scripts and the `$DIALPLAN_TYPE` phone type variable instead of dialplan-generating functions and `generateDialplan()` in provider. `generateDialplan()` (and related functionality) becomes deprecated and will be removed in a future version. (This was necessary in order to be able to trigger "some phones don't have an up-to-date configuration" notifications when dialplan changes.)

2.2.0 RC1

- Added support for module archives where are the files are in one toplevel directory.