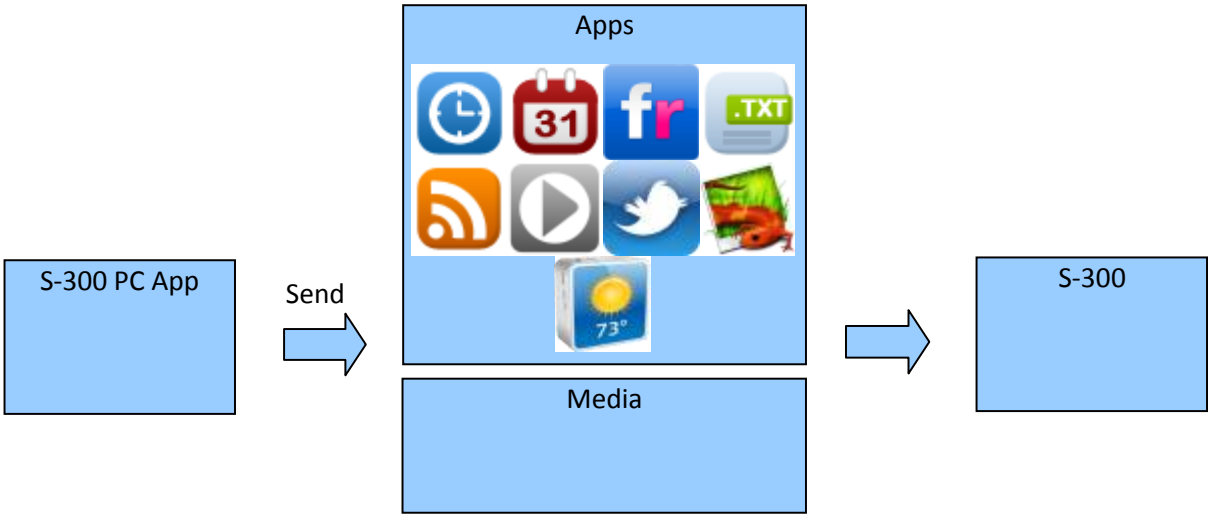
Digital Signage
App Development Guide
v1.0.2

Table of content

Platform Overview



S-300 PC Application manages all the apps imported into it. Base on user configuration, All the Apps, media files, and configuration data files will be sent to the device via FTP to S-300 devices.

- All Selected media in the playlist will be sent to the device and stored in a folder named Media, included all video, picture and audio selected by user for each app without duplication(Any same file name media files will be sent only once).
- All App information that is include in the flash folder contain in the app packaging(will be further discuss in App packaging section) will be sent along with the App SWF to the device.

Application Development

App Entry Point

For the S-300 to signal the app to start and stop, the app has to define two functions in the stage which are **appStart** and **appStop**.

```
function appStart(dataPath:String, property:Object)
{ … }
function appStop()
{ … }
```

The **appStart** function is used to inform the app to start its operation. The **appStart** function receives two arguments, **dataPath** and **property**. **dataPath** is the path to the json configuration file. The json configuration file contains extra properties defined by the developer in the app descriptor, and will be discussed in more detail in the next section. The **property** object contains two fields which are **property.width** and **property.height**. These control the width and height of the app.

The **appStop** function does not contain any arguments, it only notifies the app to clean up and release all resources occupied when switching layouts.

App Descriptor

The Digital Signage S-300 app must have an XML descriptor which provides the author details as well as app behaviour in the editor. The app descriptor contains two main sections which are **detail** and **custom** (short for customization). The general format of an app descriptor will be as follows:

```
<app>
    <detail>...<./detail>
    <custom>...</custom>
</app>
```

App Detail

Each app can specify the app author description in XML which will be shown on the editor. The elements of app **detail** are:

- name – Name of the application
- fileName – The SWF filename for the app which is stored in the flash folder in the app packaging
- thumbnail – The filename for the app icon
- description – The app’s description
- version – The version of current release
- releaseDate – The date of the app release
- publisher – The organization name which the app belongs to
- country – The country of origin for the app developer
- resize – Specify whether users can resize the app (default is **true** if not specified)

- aspectRatioResize – Specify whether the app’s aspect ratio is fixed (default is false).
- width – specify the initial width of app when added to editor's stage
- height – specify the initial height of app when added to editor's stage
- repeat – specify whether the app can have more than 1 in editor's stage(default is true)
- swapDepth – specify whether app is depth can be swap, app with swapdepth false should not show any UI(default is true)
- minWidth – overwrite the default minimum width app can have (default is 50)
- minHeight – overwrite the default minimum height app can have (default is 50)
- onAddMessage – A message to be shown on a popup box when app is added to the PC app stage

The result will be as follow:

```
<detail>
    <name>AppName</name>
    <fileName>AppName .swf</fileName>
    <description>App Description</description>
    <version>1.0.0</version>
    <release>01/01/2012</release>
    <publisher>Publisher name</publisher>
    <changeLog>-</changeLog>
    <thumbnail>AppName.png</thumbnail>
    <country>US</country>
    <url>www.AppName.com</url>
    <resize>false</resize>
    <aspectRatioResize>false</aspectRatioResize>
    <width>100</width>
    <height>100</height>
    <repeat>true</repeat>
    <swapDepth>true</swapDepth>
    <minWidth></minWidth>
    <minHeight></minHeight>
    <onAddMessage>Message to be shown...</onAddMessage>
</detail>
```

App Customization

There are additional properties that the developer can define. Developers can define their own custom field in the property pane in their app descriptor. UI controls such as check box, list, text field, number field, color picker and playlist are available. The format for specifying the property pane control will be:

```
<custom>
```

```
<canvas>...</canvas>
<inputs>
    <input type="..." … />
    <input type="..." … ></input>
</inputs>
</custom>
```

The results of the input specified in the custom tag will be generated by S-300 PC application to the json in key-value format upon project transfer. Detailed configuration of each control will be discussed in the next section

Check Box

The **check box** control is used to return a Boolean expression depending on whether or not the box is checked or unchecked. The following are attributes specific to the **check box** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control
- type – always use “boolean” (must be lower case)
- default – specify whether the check box is checked (true) or unchecked (false)

Format: <input id="checkbox1" text="Title" type="boolean" default="true">

Return: {"checkbox1":true}

List

The **list** control is used to return and index or string depending on the user’s configuration. The following are attributes specific to the **list** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control
- type – always use “list” (must be lower case)
- return – the return string or index of item selected in the current list
- defaultListId – the initial list id (string) to be used
- default – the initial item to be selected in the current list id

Format:

```
<input id="list1" text="Title" type="list" return="string/index"
defaultListId="list_source_1" default="1">
    <lists>
        <list id="list_source_1">
            <item>item 1_1</item>
            <item>item 1_2</item>
            <item>item 1_3</item>
        </list>
        <list id="list_source_2">
            <item>item 2_1</item>
```

```
                    <item>item 2_2</item>
                    <item>item 2_3</item>
                </list>
            </lists>
        </input>
```

Return: If return value is set to string: { "list1": "item 1_1" }

If return value is set to index: { "list1":0 }

Text

The **text** control is used to return a string of text. The following are attributes specific to the **text** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control
- type – always use “text” (must be lower case)
- default – the default text string to be displayed when this field is first initialize
- isRequired (optional) – specify whether the field is mandatory
- regex (optional) – specify a validation pattern
- regexMsg (optional) – Message to be shown when input string does not match regex specified

Format: <input id="textField1" text="Link" type="text" default="http://123" isRequired="true" regex ="^(http|https)\://" regexMsg="Invalid url"/>

Return: { "textField1":"http://123" }

Number

The **number** control is used to return a numerical value. The following are attributes specific to the **number** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control
- type – always use “number” (must be lower case)
- default – the initial value to be displayed
- min (optional) – specify the minimum value in a range (inclusive)
- max (optional) – specify the maximum value in a range (inclusive)

Format: <input id="numberField1" text="Title" type="number" default="0" />

Return: { "numberField1":0 }

Color

The **color** control is used to return a hex color code in integer format. The following are attributes specific to the **color** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control

- type – always use “color” (must be lower case)
- default – the initial hex format color code to be displayed

Format: <input id="color1" text="Title" type="color" default="0xFFFFFF" />

Return: { "color1":16777215 }

Playlist

The **playlist** control is used to return a list of items for playback. The following are attributes specific to the **playlist** control.

- id – unique identifier for this control, will serve as the key in generated json output
- text – display name shown in the label field for this control
- type – always use “playlist” (must be lower case)
- video – allow user to choose all supported video formats for the S300 device
- audio – allow user to choose all supported audio formats for the S300 device
- picture – allow user to choose all supported picture formats for the S300 device
- customExt – define extra file extensions that will be supported by the app other than the video, audio and picture settings above. Eg: “FLV; MP3; JPG;”
- http – define whether or not the user can input an http link (true/false)
- rtmp – define whether or not the user can input an rtmp link (true/false)

By default, the playlist control contains only 3 columns which are row count, playlist item name and path. The playlist control allows users to define extra columns. Input types supported are: number, list and text only. Below is a sample configuration.

Format:

```
<input id="playlist1" text="Playlist" type="playlist" video="true" audio="true"
        picture="true" customext="flv;">
        <columns>
                <input id="playlist_number1" text="Number" type="number" min="1"
                        max="86400" default="5"></input>
                <input id="playlist_list1" text="List" type="list" return="string">
                        <list>
                                <item>1</item>
                                <item>2</item>
                                <item>3</item>
                        </list>
                </input>
                <input id="playlist_text1" text="Text" type="text" default="Text1" />
        </columns>
</input>
```

Return:

```
{
```

```
"playlist1":[
        {
            "name":File1.jpg,
            "path":"../../media/File1 .jpg",
            "playlist_number1":5,
            "playlist_list1":"1",
            "playlist_text1":"Text1"
        },
        {
            "name":File2.jpg,
            "path":"../../media/File2 .jpg",
            "playlist_number1":5,
            "playlist_list1":"1",
            "playlist_text1":"Text1"
        }
    ]
}
```

eventInput and eventCommand

Some components will have a relationship with other components. For example, a country list can associate with a state list. For such cases, the developer can define the event handling in the component that will trigger the event. An event such as enabling or disabling a control can be done by defining the **eventInput** and **eventCommand** tag. A sample structure would be:

```
<input id="..." type="...">
    <eventInput>
        <input id="..." type="..." />
        <input id="..." type="..."></input>
    </eventInput>
    <eventCommand>
        <command if="condition">
            <input id="..." [action] />
        </command>
        <command if="condition">
            <input id="..." [action] />
        </command>
    </eventCommand>
</input>
```

The first input tag is the parent which could contain a single or multiple child control. All child components are defined within the **eventInput**. In order for a parent control to alter the child

control state, the child component must be defined within the parent's **eventInput**. Each command can have multiple child components, but each child can only belong to one parent. Only one command will be triggered at a time if the condition is met. A condition is simply a value contained in the parent control. The action can either be enable="true/false" or call a listid.

A child component can also have their own **eventInput** and **eventCommand** tags, making a child component a parent of another component. Take note that when a component is disabled, all of the child components will also be disabled.

You can only use **eventInput** and **eventCommand** in the **check box** control or the **list** control. **EventInput** and **eventCommand** will not work in controls define within playlist controls's column.

Use Case

Let's say we want to develop a video player that supports a local playlist and an online channel list. We want to have a check box that allows the user to specify whether they want to watch local or online content. We'll need a playlist component for local content and 2 lists to allow the user to select which country and which channel they want to show for online content. A sample configuration would be:

```
<inputs>
    <input id="isLocal" type="boolean" text="Local Content" default="true">
        <eventInput>
            <input id="localPlaylist" type="playlist" text="Playlist" />
            <input id="country" type="list" text="Country" return="string">
                <lists>
                    <list>
                        <item>Country1</item>
                        <item>Country2</item>
                    </list>
                </lists>
                <eventInput>
                    <input id="channel" type="list" text="Country"
                    return="index">
                        <lists>
                            <list id="c1">
                                <item>C1_Channel1</item>
                                <item>C1_Channel2</item>
                                <item>C1_Channel3</item>
                            </list>
                            <list id="c2">
                                <item>C2_Channel1</item>
                                <item>C2_Channel2</item>
                                <item>C2_Channel3</item>
                            </list>
```

```
                                    </lists>
                                </input>
                            </eventInput>
                            <eventCommand>
                                <command if="Country1">
                                    <input id="channel" listId="c1" />
                                </command>
                                <command if="Country2">
                                    <input id="channel" listId="c2" />
                                </command>
                            </eventCommand>
                        </input>
                    </eventInput>
                    <eventCommand>
                        <command if="true">
                            <input id="localPlaylist" enable="true" />
                            <input id="country" enable="false" />
                        </command>
                        <command if="false">
                            <input id="localPlaylist" enable="false" />
                            <input id="country" enable="true" />
                        </command>
                    </eventCommand>
                </input>
            </inputs>
```

SearchXML

For App that consume any API provider by web server, some case it is not posible to allow user to configure the app given that some required property is generate dynamically during runtime. For instances, Search for a location where multiple location will be return by the server. In this case the SearchXML controls allow developer to define a selection panel to allow user to select desired location. The following attibutes are specific to the **SearchXML** control.

- id - unique identifier for this control, will serve as the key in generated json output
- type – always use “search-xml” (must be lower case)
- text – display name shown in the label field for this control
- isRequired - specify whether the field is mandatory(true/false)
- externalSWF – specify whether require external SWF file name which will be called to get string or parse xml returned(Note: If defined, the SWF will be load from the root of app packaging)
- isMultiSelect – specify whether user can select multiple result(true/false)

Sample configuration is layout as below

```
<input id="searchLocation" type="search-xml" text="Result" isRequired="true" externalSWF="URL.swf" isMultiSelect=”false”>
    <url local="false">
        <param externalFunction="getRootPath"></param>
        <param value="SearchString="></param>
        <param id="searchKey"></param>
    </url>
    <list node="locations.location">
        <column title="Location Name" e4x="@cityname" />
        <column title="Country Name" e4x="@countryname" />
        <column title="State Name" e4x="@statename" />
    </list>
    <result e4x="@cityname">
        <value id="cityName" e4x="@cityname"/>
        <value id="cityCode" e4x="@citycode"/>
        <value id="zipCode" e4x="@zipcode"/>
    </result>
</input>
```

SearchXML control configuration is devided into 3 section which is **URL**, **XML result display** and **Result return**.

URL

URL section allow developer to specify how the request URL will be. The XML can be located either online or local(will be locate from root of app packaging). The local attribute in url node is a boolean base param to notify PC app to load the XML from App directory(true) or online(false).

There are 3 type attribute can be define in param tag, which is externalFunction, value and id.

- externalFunction – function name to be call in the externalSWF which will return a string(Useful to hide the path with developer api key) (Note: Will be used only when the externalSWF is defined)
- value – any static string which to be append to the end of previous param
- id – Id of any component in the property grid where the component value will be append to previous param

For URL which will be different with different setting on the property grid, the cond tag can be used to define those condition. Sample configuration is layout as below

```
<param>
    <cond componentId1=”value1” componentId2=”value2”>
        <param id=”componentId3” />
    </cond>
    <cond componentId1=”value3” componentId2=”value4”>
        <param id=”componentId4” />
        <param>
            <cond … >
                …
            </cond>
        </param>
    </cond>
</param>
```

Each cond(short for condition) tag act as a node in a search tree, each node can be branch out N time. Once a cond tag is valid, subsequence cond tag will be ignore(think it as a if else statement).

XML result display

After retrieve of XML, list tag can be use to define what information to extract from the retrieved XML and display in the DataGrid.

```
<list node="locations.location">
    <column title="Location Name" e4x="@cityname" />
    <column title="Country Name" e4x="@countryname" />
    <column title="State Name" externalFunction=”parseState” />
</list>
```

Attribute “node” define what part of xml will be extract(Must be a list of xml node) in e4x format. After define the list of xml node, column tag used to extract information from each xml node and display in different column. Attritube “title” define table header of each column. Either e4x or externalFunction can be used to extract information from each node. An XML will be pass to the externalFunction a **string** will be expect to be return from the function.

Result return

After user select on the list, data that require to be extract from node and return to app can be

configure as below:

```
<result e4x="@cityname">
    <value id="cityName" e4x="@cityname"/>
    <value id="cityCode" e4x="@citycode"/>
    <value id="zipCode" e4x="@zipcode"/>
</result>
```

Attribute e4x in the result tag define what information to be shown on the text field, externalFunction can also be used where the XML node will be pass to the function and a string will be expect to return. If it does not being define, a raw json string will be shown instead. Value tag inside result define what information will be return to the app json file. Attribute id in the value tag define what key to retrieve the value and e4x or external function could be used to define what value to be retrieve from the XML node. The output for configuration above is shown below:

```
“searchLocation” : {
    “cityName”:”My City Name”,
    “cityCode”:”My City Code”,
    “zipCode”:”My Zip Code”
}
```

If multi selection is enabled, the data returned will be an array of object instead of single object.

```
“searchLocation” : [ {
    “cityName”:”My City Name”,
    “cityCode”:”My City Code”,
    “zipCode”:”My Zip Code”
    },
    ...
]
```

eventInput and eventCommand

Some components will have a relationship with other components. For example, a country list can associate with a state list. For such cases, the developer can define the event handling in the component that will trigger the event. An event such as enabling or disabling a control can be done by defining the **eventInput** and **eventCommand** tag. A sample structure would be:

```
<input id="..." type="...">
    <eventInput>
        <input id="..." type="..." />
        <input id="..." type="..."></input>
    </eventInput>
    <eventCommand>
        <command if="condition">
            <input id="..." [action] />
        </command>
        <command if="condition">
            <input id="..." [action] />
        </command>
    </eventCommand>
</input>
```

The first input tag is the parent which could contain a single or multiple child control. All child components are defined within the **eventInput**. In order for a parent control to alter the child control state, the child component must be defined within the parent's **eventInput**. Each command can have multiple child components, but each child can only belong to one parent. Only one command will be triggered at a time if the condition is met. A condition is simply a value contained in the parent control. The action can either be enable="true/false" or call a listid.

A child component can also have their own **eventInput** and **eventCommand** tags, making a child component a parent of another component. Take note that when a component is disabled, all of the child components will also be disabled.

You can only use **eventInput** and **eventCommand** in the **check box** control or the **list** control. **EventInput** and **eventCommand** will not work in controls define within playlist controls's column.

Use Case

Let’s say we want to develop a video player that supports a local playlist and an online channel list. We want to have a check box that allows the user to specify whether they want to watch local or online content. We’ll need a playlist component for local content and 2 lists to allow the user to select which country and which channel they want to show for online content. A sample configuration would be:

```
<inputs>
    <input id="isLocal" type="boolean" text="Local Content" default="true">
        <eventInput>
            <input id="localPlaylist" type="playlist" text="Playlist" />
            <input id="country" type="list" text="Country" return="string">
```

```
                <lists>
                    <list>
                        <item>Country1</item>
                        <item>Country2</item>
                    </list>
                </lists>
                <eventInput>
                    <input id="channel" type="list" text="Country"
                    return="index">
                        <lists>
                            <list id="c1">
                                <item>C1_Channel1</item>
                                <item>C1_Channel2</item>
                                <item>C1_Channel3</item>
                            </list>
                            <list id="c2">
                                <item>C2_Channel1</item>
                                <item>C2_Channel2</item>
                                <item>C2_Channel3</item>
                            </list>
                        </lists>
                    </input>
                </eventInput>
                <eventCommand>
                    <command if="Country1">
                        <input id="channel" listId="c1" />
                    </command>
                    <command if="Country2">
                        <input id="channel" listId="c2" />
                    </command>
                </eventCommand>
            </input>
        </eventInput>
        <eventCommand>
            <command if="true">
                <input id="localPlaylist" enable="true" />
                <input id="country" enable="false" />
            </command>
```

```
                <command if="false">
                    <input id="localPlaylist" enable="false" />
                    <input id="country" enable="true" />
                </command>
            </eventCommand>
        </input>
</inputs>
```

Canvas Event

There are some cases where an app's custom property needs to change the width and height of the app. This can be accomplished by defining the canvas event (this is typically required by apps which are not resizable and the width and height value are dependent on the property configuration). An example of the structure would be:

```
<canvas>
    <cond [input id]=[value]  width="..." height="..." />
    <cond [input id]=[value] [input id]=[value] ... width="..." height="..." />
</canvas>
```

The cond (condition) tag is defined within the canvas tag. All inputs matching the criteria will be in an id-to-value format as attributes in the cond tag. Once conditions are met, the width and height attribute values are added to the default app width and height values. If the default width and height values are not specified then you would add the attribute values to default width 50 and default height 50. Each key and value within the cond tag have an **AND** relation meaning that all conditions must be met and all values will be applied. The maximum value for width is 1280 and height is 720.

Use Case

Let's say we have a clock app which can be either analog or digital depending on the user's configuration. The dimensions for the analog clock are 100x100 pixels, whereas the digital clock is 100x50 pixels. In this example we would want to allow the user to select either analog or digital but not allow them resize the app itself. When the user selects the analog clock we add 50 pixels to the height of the app. In order to accomplish this we use the following:

```
<app>
    <detail>
        ...
        <minWidth>100</minWidth>
        <minHeight>50</minHeight>
        <resize>false</resize>
        ...
    <detail>
    <custom>
        <inputs>
            <input id="clockType" text="Type" type="list" return="string">
                <lists>
```

```
                                    <list>
                                        <item>Analog</item>
                                        <item>Clock</item>
                                    </list>
                                </lists>
                            </input>
                        </inputs>
                        <canvas>
                            <cond clockType="Analog" height="50">
                        </canvas>
                    </custom>
                </app>
```

App Preview

In order for S-300 PC App to signal Apps for preview session in PC App. Apps are required to specified 2 function in the stage as shown in below:

```
function previewStart(dataPath:String, prop:Object, previewObj:Object):Void
{
	...
}

function previewStop():Void
{
	...
}
```

S-300 PC App will called the previewStart function to signal the app for a preview session. Arguments passed in are listed as below:

1. **dataPath** is the path where the json configuration file is stored.
2. **prop** contain 2 property which is prop.width and prop.height which inform the app the dimension of the app.
3. **previewObj** contain 1 property which is previewVideoPath. For those app that play media format only supported by our device, can use this video for the preview session due to flash only support FLV and MP4 video format only.

Apps that does not define these function will be treated as unsupported and a icon with cross will be shown to indicate the app does not support preview.

Once the user exit the preview session, S-300 PC App will called the previewStop function to signal the app to clean up all resources used and exit the preview.

Packaging

For an app to successfully be imported into the S300 PC software, the following structure is required.

1. AppName (directory)
    1. AppName.xml (App Descriptor)
    2. AppName.jpg / AppName.png
    3. flash (directory)
        1. AppName.swf
        2. … (any extra files required that will be included during project transfer)

The directory name of the app packaging must be the same as the app descriptor name to allow the PC software to locate the descriptor file. The app icon and app swf file name must be same name defined in the app descriptor. Any extra resource files that are required to be transferred together with the app can be included in the flash directory (Note: extra resources should be kept as small in size as possible).

Loader Custom Event

For Apps that aggregate media playback API listed in S-300 DavidBox SDK API.pdf file. Apps can listen to the event fired in the device via the _root.addCSEventListener function. All event is listed as below

An event object contain specify information about the event will be pass to the function added into the _root.addEventListener. Structure as below:

```
_root.addCSEventListener("event key", eventHandler);
function eventHandler(evt:Object):Void
{
    var type:Object = evt.type
    ...
}
```

API	Event key	Event Object property	Description
start_vod / start_aod / insert_vod_queue / insert_aod_queue	playback start	• type :String - audio/video	Indicate the device was start playback
	playback end	• type :String – audio/video • error :String – string indicate the error detail.(Listed in Playback error code section)	Indicate the device was ended current playing media files
	Playback terminate	• type:String - audio/video	Indicate the device was exited video playback
start_pod / insert_pod_queue	slideshow start	N/A	Indicate the device was start picture playback
	slideshow next	N/A	Indicate the device was start transition of next picture
	slideshow end	N/A	Indicate the device was transition out current picture
	photo playback terminate	N/A	Indicate the device was exited picture playback

Playback Error code

Error String	Description
0	NO_ERROR
1	INVALID_FILE
2	OPEN_FILE_ERROR
3	READ_FILE_ERROR
4	CONNECTION_ERROR
5	DETECTION_ERROR
6	DISC_ERROR
7	DISC_REGION_ERROR
8	SYSTEM_ERROR
9	QUEUE_ERROR
10	NETWORK_ERROR
12	FORMAT_NOT_SUPPORTED
13	ERROR_UNKNOWN

Serial Port Support

The S-300 supports a serial port communication which enables developers to develop interactive apps. External devices can send binary data through the serial port to device which will be then received by the app and perform any necessary actions. The serial port configuration as below:

1. Bits per second/baudrate : 115200
2. Data bits : 8
3. Parity : None
4. Stopbits : 1

The message must be encoded in ASCII format, following binary format is required in order for the app to receive the raw string data

[ack][length][data]

➔ ack (1 byte) – indicate start of message (0xFF)

➔ length (2 bytes) – the length of message (min : 1, max : 1023)

➔ data - ASCII character in range from (0x20 – 0x7F), message length according to length specified

Length Calculation

To avoid conflict with ASCII special characters ($0x00_8$ - $0x1F_8$), the length specified in the message has to be custom handled. The length received will be split by 6 bits and each portion will be applied to binary operation **OR** with $0x40_8$ (01000000_2)

Min:

1 = 0x001 = 000000 000001

00000000 | 00000001

OR 01000000 | 01000000

01000000 | 01000001

0x40 | 0x41

Max:

1023 = 0x3FF = 1111 111111

00001111 | 00111111

OR 01000000 | 01000000

01001111 | 01111111

0x4F | 0x7F

For example, we would like to send hello world

"Hello world" = 11 char = 11 bytes

11 = 0x0B = 0000 1011

00000000 | 00001011

OR 01000000 | 01000000

01000000 | 01001011

0x40 | 0x4B

So the length would be 0x40 0x4B = 0100000001001011

App Receiving the Message

Once the app is loaded, the app can add a callback function to receive the serial message as below

```
_root.addCSEventListener("serial", this.onSerialEvent);
function onSerialEvent(evt:Object):Void
{
    var data:String = evt.data;
    ...
}
```

Any message sent through the serial port can be received by the callback and the app can do any processing the subject needs.

Tips

Serial communications are prone to interference; the maximum length of serial cables varies depending on the surrounding environment. Please refer to the following links for reference.

http://www.activexperts.com/serial-port-component/tutorials/pinout232/

http://www.himatix.com/resources/tech/serial.html

External Loader

S-300 allow developers to load their own Flash SWF file which will overwrite the existing loader in the device. Developer can specify the path to their loader by creating a file named “loader.json” and put it into the device's root directory and configure as below:

```
{
        "externalLoaderPath":"the path to your loader"
}
```

The path can be any http path or local path. The path to the pendrive root directory is "/opt/sybhttpd/localhost.drives/USB_DRIVE_DSS/"

Note:Once the loader has being overwritten, all scheduler and FTP ability via S-300 PC application will be lost.