Integration Guide for third-party applications into RDK-B stack

Overview

This guide is intended to provide guidance for those who want to integrate their custom application to an RDK-B based device. The guide covers the below cases

- How to add a third-party application to the Yocto build system
- How to get the application running in the target device
- How to control and configure the application
- How the application can fetch data from the platform to share to head-end

The guide is based on the Raspberry Pi reference platform available for community, but is easily portable to other devices with minimal changes. The below mentioned names are used for the purpose of explaining the process in the guide (names for each are kept slightly different for ease of distinguishing in between)

Application name:	**sampleAppn**
Source file:	sampleapp.c
Service name:	sample
Meta layer folder:	meta-rdk-sampleapp
Recipe name:	sampleapp

Integrate application to Yocto build system

RDK is based on Yocto build system. Based on Yocto philosophy, it is advised to use a separate layer for the sample application which will provide better portability also. A sample folder with desired structure is created and attached as zip. Please refer it for examples

Steps to be followed:

Create a new meta layer for storing build related data

In order to add the new application to the Yocto build system (and to keep it independent of existing RDK versions as well as SoC/OEM/MSO), add a new meta layer. For this, a new folder named meta-<layername> will be added in the base directory of the Yocto build. The layer details need to be mentioned in layer.conf file.

In the example used

a) Create a folder named meta-rdk-sampleapp for keeping the layer configuration files

$ ls meta-rdk-sampleapp

conf/ licenses/ LICENSES.TXT recipes-new/

b) Add the layer specific configurations to a conf file, so that it can be taken up by the build system when parsing for recipes. The layer configuration is stored in a layer.conf under the conf folder in the meta layer folder.

The configuration file contains settings to take all the bitbake and bitbake append recipes inside the meta-layer during the build, as well as some Yocto related generic configs required for a layer config file.

$ ls meta-rdk-sampleapp/conf/layer.conf

meta-rdk-sampleapp/conf/layer.conf

c) Mention the licenses file. Apache 2.0 License file is the default license. Other licenses also can be included (provided related changes are done in recipe files)

$ ls meta-rdk-sampleapp/LICENSES.TXT

meta-rdk-sampleapp/LICENSES.TXT

$ ls meta-rdk-sampleapp/licenses

Apache-2.0

Create a new bitbake recipe file for the application

Add a bitbake recipe file that provides the details of the application including location of code, any config files to be imported, any variables to be set (w.r.t Yocto build system) etc.).

Add relevant fields like **LICENSE,LIC_FILES_CHKSUM, inherit** etc.

In the example, a bitbake recipe is added at the below mentioned location (so as to accommodate applications with multiple recipes associated with them across various inner layers):

$ ls meta-rdk-sampleapp/recipes-new/sampleapp/*.bb

meta-rdk-sampleapp/recipes-new/sampleapp/sampleapp.bb

License used is Apache 2.0 and autotools is inherited

```
LICENSE = "Apache-2.0"
S = "${WORKDIR}/sample"
LIC_FILES_CHKSUM = "file://LICENSE;md5=175792518e4ac015ab6696d16c4f607e"
inherit autotools
```

Add/Point to the source code(along with build files) in the recipe

Once the bitbake recipe file is added, point to the source file location from within the bitbake recipe. This can be achieved with the help of **SRC_URI** variable. If multiple sources are to be fetched, **SRC_URI +=** will do the trick

In the example, a tar ball is used as the source location for the source code. So the location where the tar ball is present (using **FILESEXTRAPATHS_prepend** variable) needs to be specified

```
FILESEXTRAPATHS_prepend := "${THISDIR}:"
SRC_URI = file://sampleapp.tgz
```

Add a service file to control the application

In order to control the application, it should be registered as a service. This will help systemd to take care of starting/stopping the service.

In the example, the service file is also added inside the tar ball. The location of service file is within a scripts folder (for ease of separating it from the build related files) and is at the below location

$ ls meta-rdk-sampleapp/recipes-new/sampleapp/files/sample/scripts/sample.service

meta-rdk-sampleapp/recipes-new/sampleapp/files/sample/scripts/sample.service

The service file just mentions the location of the sample application (which needs to be controlled by it) along with some generic service file information

$ cat meta-rdk-sampleapp/recipes-new/sampleapp/files/sample/scripts/sample.service[Unit]

Description=Example sampleapp systemd service.

[Service]

Type=simple

ExecStart=/usr/bin/sampleAppn

ExecStop=/bin/sh -c ' echo "Sample service Stopped" '

[Install]

WantedBy=multi-user.target

Define the deliverables of the recipe

Deliverables of the recipe includes the files that will be packaged and included in the final binary, as part of this component. This might include executable, script files, xml/con files, certificates etc.

In the example, the deliverables are the sample application binary (**sampleAppn)** and the service file (**sample.service**) . The application binary will be taken up by the recipe as part of its default behaviour.The service file, should be manually copied to rootfs using **do_install_append** function, as well as marked as a service file using the **SYSTEMD_SERVICE_${PN}** variable. Both the service file and the binary should be marked for packaging to the final image too using **FILES_${PN}** variable. The below mentioned additions in the sample app bitbake recipe file should do the trick

```
do_install_append () {
   # Config files and scripts
   install -d ${D}${systemd_unitdir}/system
   install -D -m 0644 ${S}/scripts/sample.service ${D}${systemd_unitdir}/system/sample.service
}
SYSTEMD_SERVICE_${PN} += "sample.service"
FILES_${PN} += " \
     /usr/bin/* \
       ${systemd_unitdir}/system/sample.service \
"
```

Add the meta-layer to the machine configuration, so that this is taken up during compilation

The changes to add the recipe to the build configuration for the target is device specific (ie the target device to which the application is being integrated).

This generally involves (but might differ from platform to platform) adding the layer to the setup-environment file and to add the bitbake recipe name to the main package group recipe.

In the example, the below changes are added to the respective files for raspberry pi reference platform build system,

- In **meta-cmf-raspberrypi/conf/layer.conf**

  LAYERDEPENDS_cmf-raspberrypi = " rdk-sampleapp"

- In **meta-cmf-raspberrypi/setup-environment**

  # Add meta-rdk-sampleapp only if not already present for RDK-B

  echo "${_RDK_FLAVOR}"

  if [["${_RDK_FLAVOR}" = "rdkb"]]

  then

```
if [ $(grep '^BBLAYERS' conf/bblayers.conf | grep -c 'meta-rdk-sampleapp') -eq 0 -a -d  $TOP_DIR/meta-rdk-sampleapp ]
then
    cat >> conf/bblayers.conf <<EOF
BBLAYERS =+ "\${RDKROOT}/meta-rdk-sampleapp"
EOF
  fi
fi
```

- In **meta-cmf-raspberrypi/recipes-core/packagegroups/packagegroup-rdk-oss-broadband.bbappend** , add this to the end of **DEPENDS_packagegroup-rdk-oss-broadband_append** variable.

```
sampleapp \
```

Run the application in target device

This section details how the application can be run as a service. Running the application as a cron job or any other method standard to Linux is not included here (but can easily be achieved following standard Linux practices). Running the application by a trigger from head-end is explained later in section Control & Configure the application

As mentioned in previous section, the application is packaged along with a service file that can be used to start/stop/restart it as per standard systemd commands.

- To start:

  systemctl start <servicename>

- To stop:

  systemctl stop <servicename>

**Note that any logic that needs to be run before exit of the service needs to be taken care externally (see below section Control & Configure the application for an example) as systemd won't be able to handle it*

- To restart:

  systemctl restart <servicename>

In the Raspberry Pi example, the name of the service is sample, so the commands to start, stop or restart the service will be (in the order)

- To start:

  systemctl start sample

- To stop:

  systemctl stop sample

- To restart:

  systemctl restart sample

To check the status of the service, the command **'systemctl status sample'** can be used

Auto start the application

This section details how the application is set to start/stop automatically during boot up. If no other special control is required to start/stop the service and it can just start at boot up of device, it can be simply achieved by marking the service as enabled against the **SYSTEMD_AUTO_ENABLE** variable in the bitbake recipe. The change would look as given below:

SYSTEMD_AUTO_ENABLE = "enable"

In the Raspberry Pi example, this needs to be added anywhere in the file **meta-rdk-sampleapp/recipes-new/sampleapp/sampleapp.bb**

In a real world scenario, the application start-up will be mostly conditional. To achieve this, the sys config variables available within the RDK-B can be used. The Syscfg system usually consists of a persistent data base which will be used up by the RDK-B to maintain status/data of various parameters. Syscfg offers a set operation (which can be used to update the status) and a get operation (which can be used to retrieve the current status). In order to achieve conditional start, the below two changes are required

1. The default value of the variable needs to be added to the system_defaults file (this will be used up during first boot as well as boot up following a factory reset). This file will be used to populate the Syscfg db during first time boot

The file is present in /etc/utopia/system_defaults in the CPE. In order to add the variable and its default value, the file system_defaults need to be added with the variable name and default value

In the Raspberry Pi example, the system_defaults file is present at **meta-cmf-raspberrypi/recipes-ccsp/util/utopia/system_defaults.** The below line should be added to the file (1 denotes enabled and 0 denotes disabled in the example)

sampleapp_enabled=1

2.The regular value of the variable needs to be fetched during start-up so as to decide if service needs to be started or not.

The regular file is usually present in the /nvram/syscfg.db file in CPE. Once the file is populated, there is no change required to update the values. To set/get the value in syscfg.db, the below commands can be used

- To set value to 0

syscfg set sampleapp_enabled 0

- To set the value to 1

syscfg set sampleapp_enabled 1

- To save the value to Syscfg db file, the below command needs to be run

syscfg commit

- To retrieve the value

syscfg get sampleapp_enabled

3. The variable value needs to be used to control service start-up

Instead of starting the application as such(from the service file), a conditional check needs to be added in the service file which gets the value of the sysconfig variable sampleapp_enabled and then starts the sample application only if the value is set to 1. In the example, the line in black needs to be changed to the line in brown in the file **meta-rdk-sampleapp/recipes-new/sampleapp/files/sample/scripts/sample.service**

ExecStart=/usr/bin/sampleAppn

ExecStart=/bin/sh -c 'val=`syscfg get sampleapp_enabled`; echo "$val"; if ["$val" == 1]; then /usr/bin/sampleAppn; fi '

To verify the results, set the value of ‘sampleapp_enabled’ variable to 1, restart the device and check the status of service using the command **‘systemctl status sample’**

Control/Configure the application

The standard method to control and configure an application or component in RDK-B / CCSP is to use TR-181 data models. This helps the application to communicate with other CCSP components as well as head end systems using the standard communication mechanism in RDK-B.

Depending on the requirements, the data models can be part of

1. An existing CCSP component (for e.g.: CcspPandM) or

2. A new CCSP Component created for ad-hoc control of the sample application

Adding a New Test Component:

This section briefs how to add a new component along with TR-181 data models to RDK-B.

Test Component API List

API Name	Description
• TestComponent_GetParamUlongValue	API is used to retrieve the parameter value of type “unsignedInt”
• TestComponent_SetParamUlongValue	API is used to set the parameter value of type “unsignedInt”
• TestComponent_GetParamStringValue	API is used to retrieve the parameter value of type “string”
• TestComponent_SetParamStringValue	API is used to set the parameter value of type “string”
• TestComponent_Commit	To Commit all the update to the data model

Build ccsp-testcomponent for Raspberry Pi device

1. Put the ccsp-testcomponent.bb recipe file inside meta-rdk-broadband/recipes-ccsp/ccsp location.
2. Create folder names "files" under "meta-rdk-broadband/recipes-ccsp/ccsp" path
3. Put CcspTestComponent.tar.gz (source code of TestComponent) file inside meta-rdk-broadband/recipes-ccsp/ccsp/files location.
4. Build the test component inside build directory i.e. build-raspberrypi-rdk-broadband directory.
   bitbake ccsp-testcomponent
5. Binary file of test component will be present inside build-raspberrypi-rdk-broadband/tmp/work/cortexa7hf-neon-vfpv4-rdk-linux-gnueabi/ccsp-testcomponent/1.0-r0/image/usr/bin/ location
6. XML file of test component will be present inside build-raspberrypi-rdk-broadband/tmp/work/cortexa7t2hf-neon-vfpv4-rdk-linux-gnueabi/ccsp-testcomponent/1.0-r0/image/usr/ccsp /testcomponent/ location.

NOTE : For rpi4 , yocto 3.1 use ccsp_testcomponent.tar.gz to avoid build errors due to cfg directory missing

Run the Test Component

Copy the supported files to Raspberry Pi device

Copy test_component (Binary of ccsp-testcomponent, generated in step 4 of Build) and TestComponent.xml (present at location mentioned in step 5 of Build) to /tmp/ directory of Raspberry Pi device.

Execute the TestComponent in the Raspberry Pi device

Go to /tmp/ directory of Raspberry Pi 3 and start the ccsp-testcomponent
./test_component -subsys eRT.

You can check the status of the process by doing *ps* in the RPi device.

Validate if component is registered with CR

DMCLI (Database Manager Command Line Interface) provides interface used to send and receive command/messages via CLI (Command Line Interface) over Telnet and SSH protocols.

List all the test component parameters using dmcli:

dmcli eRT getv Device.TestComponent.

```
Expected Output:
CR component name is: eRT.com.cisco.spvtg.ccsp.CR
subsystem_prefix eRT.
getv from/to component(eRT.com.cisco.spvtg.ccsp.testcomponent): Device.TestComponent.
Execution succeed.
Parameter    1 name: Device.TestComponent.TestSampleParamUlong
               type:       uint,    value: 1
```

Change the "TestSampleParamUlong" parameter value by using below command:

dmcli eRT setv Device.TestComponent.TestSampleParamUlong uint 5

Now verify the value by this command

dmcli eRT getv Device.TestComponent.TestSampleParamUlong

Include Newly Added Component in Package Group

Once the validation is successfully done and we can now include the new component (ccsp-testcomponent) in the package groups.

1. Open "packagegroup-rdk-ccsp-broadband.bb" file present at "meta-rdk/recipes-core/packagegroups/packagegroup-rdk-ccsp-broadband.bb" location.
2. In the "RDEPENDS_packagegroup-rdk-ccsp-broadband" flag add the name of new component.

Adding a new parameter to the existing component

This method can be utilized if the sample application has functionality similar to an existing component or if there are no detailed configurations required to be done from head-end using different protocols (which are abstracted by RDK-B and converter to TR-181 format). In this options, a new TR-181 data model can be added to an existing component. To achieve this, the changes required are

- Add the data model details in the corresponding xml file
- Add the data model parameter to the corresponding DML file
- Add the underlying functionality to the corresponding API file

Refer files in folder sampleDatamodel_Existing_Component for more details.

Adding the data model details in XML file

A data model is known to exist, along with its characteristics, based on its configuration in xml file. In order to add the details in XML, the first thing to check is whether the type of the data model (Boolean, int , string etc.) is already supported in the targeted object family. If it is already present, skip to next section where adding data model parameter is explained. If it is not present, need to add the set and get function details to xml file

<object>

*<name>**X_CISCO_COM_DeviceControl**</name>*

<objectType>object</objectType>

<functions>

*<**func_GetParamTypeValue>GetFunctionName_For_Type</func_GetParamTypeValue**>*

*<**func_SetParamTypeValue> SetFunctionName_For_Type </func_SetParamTypeValue**>*

In the above code, replace the highlighted text within <> with proper data type. Also, replace the function name with proper get and set function names for the type as defined in DML file in next section

In order to add the details of data model, add the below code block within the **<parameters> </parameters>** block. Syntax should be

<parameter>

*<name>**DataModel_Name<**/name>*

*<type>**DataModel_Type**</type>*

*<syntax>**DataModel_Type_Syntax**</syntax>*

*<writable>**True_If_DataModel_Is_Writable_False_If_ReadOnly**</writable>*

```
</parameter>
```

In the above snippet, replace highlighted code with respective values (use existing data models for a reference, if required)

In the example case, the data model will be added under **Device.X_CISCO_COM_DeviceControl**. The data model is of type bool. The set and get functions are **X_CISCO_COM_DeviceControl_GetParamBoolValue** and **X_CISCO_COM_DeviceControl_SetParamBoolValue.** So the code will look like

```
<object>
        <name>X_CISCO_COM_DeviceControl</name>
 <objectType>object</objectType>
 <functions>
<func_GetParamBoolValue>X_CISCO_COM_DeviceControl_GetParamBoolValue</func_GetParamBoolValue>
<func_SetParamBoolValue>X_CISCO_COM_DeviceControl_SetParamBoolValue</func_SetParamBoolValue>
..
..
..
<parameters>
<parameter>
<name>SampleAppEnable</name>
<type>boolean</type>
<syntax>bool</syntax>
<writable>true</writable>
</parameter>
```

Add the data model parameter to the corresponding DML file (cosa_x_cisco_com_devicecontrol_dml.*)

Any invocation of the data model results in the call being redirected to the functions defined in DML file. These are the same functions that are being mentioned in the XML file. Changes include (refer attached code files for details)

1. Declare the set and get functions (mentioned in the XML) in header file and define them in c file (Skip this if data model is of an existing type)
2. In the definition of both functions, add an 'if' case to check if the invocation is for the defined data model

```
if (AnscEqualString(ParamName, "DataModelName", TRUE))
{
…
}
```

Replace the highlighted part in above snippet with the data model name mentioned in XML

1. In the get function, either invoke the corresponding get function in API file or directly get it from syscfg APIs. In either case, update the output variable passed into the function (refer attached code files for details)
2. In the set function, invoke the corresponding set function in API file. Return status based on API function return value (refer attached code files for details)

In the example, the below code will be added to the get function **X_CISCO_COM_DeviceControl_GetParamBoolValue** , which will read the value from syscfg db and returns the value

```
if (AnscEqualString(ParamName, "SampleAppEnable", TRUE))
{
        char buf[8] = {0};
        syscfg_get( NULL, "sampleapp_enabled", buf, sizeof(buf));
        if( buf != NULL )
        {
```

```
                pMyObject->Enable=(BOOL)atoi(buf);
                *pBool=pMyObject->Enable;
                CcspTraceInfo(("syscfg get done for SampleApp\n"));
                return TRUE;
        }
        else
        {
                CcspTraceInfo(("syscfg get failed for SampleApp \n"));
                *pBool=pMyObject->Enable;
                return TRUE;
        }
}
```

For the set function **X_CISCO_COM_DeviceControl_SetParamBoolValue,** the below code snippet is used (where **CosaDmlDcSetSampleApp** is the API function for set) which will set the value in syscfg db, invokes the API function and then returns the status

```
if (AnscEqualString(ParamName, "SampleAppEnable", TRUE))
{
        char buf[8] = {0};
        pMyObject->Enable = bValue;
        snprintf(buf,sizeof(buf),"%d",bValue);
        if ( syscfg_set(NULL,"sampleapp_enabled", buf) != 0 )
        {
                AnscTraceWarning(("syscfg_set failed\n"));
        }
        else
        {
                if(syscfg_commit() != 0)
                {
                        AnscTraceWarning(("syscfg_commit failed \n"));
                }
        }
        retStatus = CosaDmlDcSetSampleApp(NULL, pMyObject->Enable);
        if ( retStatus != ANSC_STATUS_SUCCESS )
                return FALSE;
        return TRUE;
}
```

Add the API in corresponding API file (cosa_x_cisco_com_devicecontrol_apis.*)

The API file contains the actual API (invoked by the function in DML file) that performs the action intended to be performed by the SET operation (and in most cases the data fetch for the GET operation). Changes include (refer attached code files for details)

- Add the declaration of the API in header file
- Add the definition in the corresponding C file
- In the definition body, run the corresponding commands to start the service (either using system () or exec () or choice of other functions)

In the example, the below code snippet is added to start (restart is used in case service is stuck and Data Model is invoked to recover) and stop the service

```
ANSC_STATUS
CosaDmlDcSetSampleApp
(
        ANSC_HANDLE hContext,
        BOOLEAN pFlag
)
{
        if (pFlag == 0)
        {
                system("systemctl stop sample");
                CcspTraceInfo(("Inside stop\n"));
                return ANSC_STATUS_SUCCESS;
        }
        else
        {
                system("systemctl restart sample");
                CcspTraceInfo(("Inside restart\n"));
                return ANSC_STATUS_SUCCESS;
        }
        //   return ANSC_STATUS_SUCCESS;
}
```

Adding a new CCSP Component created for ad-hoc control of the sample application

This method helps in having an ad-hoc CCSP agent to control and configure the application. Advantages include lesser or no dependency on other components (including their health), portability to multiple platforms and ease of use in upgraded versions of RDK-B. In this method, a new CCSP component is added along with data models to control the application (steps to add the data model in this case is same as mentioned in above section)

The detailed sample files are included in the sample_ccspComponent folder in the associated zip file which should be referred to and used as a template to start with

In short,

1. Add the bitbake recipe file ccsp-testcomponent.bb to the meta layer where the sample component is added (see Integrate application to Yocto build system)
2. Update the bitbake recipe to point to actual code, as well as any required changes
3. Update bblayers or conf files for required changes, if any. Add the recipe to relevant package group file and update the component name in RDEPENDS flag(see Integrate application to Yocto build system)
4. Update the XML file at CcspTestComponent\scripts\TestComponent.xml with required data models to control the application. Refer section Adding a new parameter to the existing componentt for details on how to add data models
5. Update the CCSP Component service file for any required conditional start at CcspTestComponent\scripts\CcspTestComponent.service
6. Update the DML and API functionalities (for ease, both are in the same file in the attached sample) in the file at CcspTestComponent\source\TestComponent\cosa_apis_testcomponentplugin.*
7. Update the SRC_URI in the bitbake recipe to point to the location with updated code (either recipe or tgz file)
8. Makefiles etc. are already available in the tgz file with required config
9. Compile and generate image. Verify CCSP component service is running fine or not

Detailed steps are given below

Integrate the component to Yocto build system

Refer Integrate application to Yocto build system for more details on how to integrate a component in Yocto. Refer section Adding a new parameter to the existing componentt for details on how data models are added in a component

1. Put the ccsp-testcomponent.bb recipe file inside *meta-rdk-broadband/recipes-ccsp/ccsp* location.
2. Create folder names "files" under *meta-rdk-broadband/recipes-ccsp/ccsp* path (Note: In real world, this step should be replaced with fetching source code directly from git)
3. Put CcspTestComponent.tar.gz (source code of TestComponent) file inside *meta-rdk-broadband/recipes-ccsp/ccsp/files* location.
4. Re-run the 'source' and 'bitbake' commands
   bitbake ccsp-testcomponent
5. Binary file of test component will be present inside *build-raspberrypi-rdk-broadband/tmp/work/cortexa7hf-neon-vfpv4-rdk-linux-gnueabi/ccsp-testcomponent/1.0-r0/image/usr/bin/* location
6. XML file of test component will be present inside *build-raspberrypi-rdk-broadband/tmp/work/cortexa7t2hf-neon-vfpv4-rdk-linux-gnueabi/ccsp-testcomponent/1.0-r0/image/usr/ccsp/testcomponent/* location

Validate the component

1. Copy *test_component* (Binary of ccsp-testcomponent, generated in step 4 of Build) and TestComponent.xml (present at location mentioned in step 5 of Build) to */tmp/* directory of test device
2. Go to */tmp/* directory of test device and start the ccsp-testcomponent by running the command
   ./test_component -subsys eRT.
3. *Verify the component is registered and accessible by running a data model walk using below command*
   dmcli eRT getv Device.TestComponent.
4. Change the "TestSampleParamUlong" parameter value by using below command:
   dmcli eRT setv Device.TestComponent.TestSampleParamUlong uint 5
5. Now verify the value by this command
   dmcli eRT getv Device.TestComponent.TestSampleParamUlong

Include Newly Added Component in Package Group

Once the validation is successfully done and we can now include the new component (ccsp-testcomponent) in the package groups.

1. Open packagegroup-rdk-ccsp-broadband.bb file present at *meta-rdk/recipes-core/packagegroups/packagegroup-rdk-ccsp-broadband.bb* location.
2. In the *RDEPENDS_packagegroup-rdk-ccsp-broadband* flag add the name of new component.

Following this, in the next build, this component will be taken up by bitbake and packaged in the final build image

How the application can fetch data from the platform to share to head-end?

This section details on how to fetch the data for the running application. RDK-B platforms offer many methods to collect data from the CPE. This includes (but not limited to)

1. TR-181 data models
2. Linux commands
3. Log files
4. SoC/OEM specific utilities
5. RDK-B specific files

TR-181 data models

TR-181 data models are the de-facto method to fetch data from multiple components in RDK-B. As per RDK-B Architecture, HAL calls to fetch data from platform should be invoked only by the corresponding CCSP component and other components as well as 3rd party should be using TR-181 data models to fetch data from that CCSP component. The sampleapp can use data models to fetch data internally, based on request or based on a polling mechanism. There are APIs like Cdm_GetParamString(), to fetch data using available TR-181 data models.

Linux Commands

Basic Linux commands to get IP, ARP, trace route etc. can be used to fetch basic networking info as well as device health status (CPU, Memory, Load average etc.). Depending on the support from SoC, more Linux executables might be available. Utilities as part of RDK-B stack like syscfg, sysevent, psmcli etc. can also be used to fetch data

Log files

A good amount of information are available from log files. Logs are located at /rdklogs/logs/ and each component has its own log file so as to collect component specific logs. Generic logs like wifihealth.txt, ArmConsoleLog.txt, Boot log etc. have data either from multiple components or data not restricted to any components

SoC/OEM specific utilities

This depends on the support provided by SoC, but utilities like brctl etc. should be available

RDK-B specific files

Many of the device specific information are available from RDK-B specific files like /etc/device.properties, /etc/utopia/system_defaults etc. and can be used to fetch specific data

How the application can share collected data to head-end?

Depending on the data format, there exists multiple methods to share the data with head-end. This includes (but not limited to)

1. SNMP – Support already available in RDK-B
2. TR-69 – Support already available in RDK-B
3. WebPA – Support already available in RDK-B
4. HTTPS POST (JSON/file) – Post data/file using curl in secured connection
5. SFTP (file)-Post collected data in file to FTP servers(for cloud operations)
6. Custom secured connections – Custom server client direct connections

Additional Links

- Supported TR-181 data models

  RDK-B TR-181 Data Model

- Refer the below to fetch the telemetry details from device

  Telemetry - configurations , working procedure

Attached files

The sample code is attached in IntegrationGuide_SampleCode .The following table provides the details on the attached code:

Folder	Contents	Remarks
sample_application	meta-rdk-sampleapp.zip	A sample application that serves as an example for the actual sample application
sample_ccspComponent	ccsp-testcomponent.bb CcspTestComponent.tar.gz	A sample Ccsp component that can be used to control the sample application. The bb file is the bitbake recipe and the tar ball contains all the sample code for a Ccsp component and the required functionality can be achieved by making changes in the same
sampleDatamodel_Existing_Component	cosa_x_cisco_com_devicecontrol_apis.c cosa_x_cisco_com_devicecontrol_dml.c cosa_x_cisco_com_devicecontrol_apis.h cosa_x_cisco_com_devicecontrol_dml.h	Files with sample changes that help to add a TR-181 data model in an existing component(including both DML and API files