Table of Contents

WSPS Resource Management

WSPS Resource Management primarily functions as a suite of applications that provide a container deployment management API for controlling containerization applications. The management functions are built on top of Kubernetes. While out-of-the-box kubernetes package management can provide simple deployment capabilities in a production cluster, the need for the resource management suite provides a much more of a fine-grained deployment model. The reasoning is that the type of clusters are those with nodes that have physcially attached radio hardware. The nodes also run specific RF enabled software framework applications that have a very symbotic relationship with the physical radio hardware. Leveraging Kubernetes and Docker allow the software to be orchestrated containers. Because of the need for close coupling with an RF Frontend, commerical off-the-shelf kubernetes package management doesn't provide the granualrity of control at the kernel/network/host levels for the conditions for the RF enabled software applications.

The purpose of this document is explain the purpose and architecure of the Resource Managment Product Family as well as common deployment methodologies that are our WSPS best practices and standard ways of operating application deployments.

Resource Management Architecture

The Resource Management suite of applications is bound by the clear separation of resources, data, and ipc interfaces. Specifically, the logical processing element of the framework is an open source tool for building and manipulating workflow style processing. The library set is called “Workflow-Core” Workflow Core is a light weight workflow engine targeting .NET Standard. Think: long running processes with multiple tasks that need to track state. It supports pluggable persistence and concurrency providers to allow for multi-node clusters.

The workflow decision making process is currently backend by a in memory data store construct calledRedis, which will provide the SOH data storage to enable the decision process throughout the workflows implemented by the applications. An additional noSQL implementation using MongDB will be made available in the 3Q 2020.

Lastly, an integrated Protobuf and gRPC implementation provides the level of control required to define and manage all mission resources.

WSPS API Services repo contains all of the protobuf definitions utilized by Resource Management as well as other generic IPC interfaces and client implementations.

This set of protobuf API operates on concept of a "Mission Stack" (i.eMissionService). A stack represents the "Managed Resources" that can be either

"Applications" or "Hardware" Resources (i.e ResourceService). Software Application Resources provide Application SOH and Deployment Configuration Status (i.e ApplicationService) . Hardware Resources provide the "Managed" Device/Hardware SOH through the ResourceService API since the current architect anticipates Hardware Resource Management application to be mapped to a specific hardware device.

Concept of Operations

The concept surrounding RF platform/application container orchestration is a multi-application approach that focusing on the management business logic from a deployment perspective and captilizes on insulation layers to provide generic operations for communicating, data access, and the maanged endpoint/target. The following figure represents the general application architecture that all Resource Management Applications are constructed in this same manner.

This framework approach allows program/end-users the ability to take advantage of the generic interfaces and data manaagement, but limit the custom focus or requirements to manipulate the workflows managed under the Workflow-Core elements of the applications. This utimately lets each Resource Management Application adapt to any program/end-user.

The following diagram depicts the near minimal application requirements for operating Resource Management Application. The outlined black box represents the RM apps. In its purest minimal form, would require Device Management(DM) and Cluster Resource Management (CRM) applications to successfully manage deploying application only mission stacks. If hardware requirements exist the Hardware Resource Management Applications would plug into the ResourceService and utilize either a specific API implementation or utilize a workflow-core engine if the business logic for operating the hardware is complex enough to warrant that sophisticated of a solution.

The interfaces shown relate back to the Protobuf definitions described earlier in this document. And it demonstrates some of the adminstrative uses and external interfaces expected for operating in this managed environment.

The following diagram is a scaled extension of the Site Management. This scales the solution so that it can extend off the existing Mission Stack API and provide a higher magnitude of resource management across an enterprise or multi-site operations. It demonstrates a higher level of automation at the enterprise level therefore reducing adminstrator staff oversight or tasking.

These applications are depicted in the simplest forms, but these applications can be customized to meet specific needs, by maintaining this overall architecture allows for sharing and extending across program use cases.

The following diagram shows a common approach to the business logic surrounding the overall Resource Management concept. It will provide a reubric for implementing either workflow-core workflow implementations or other business logic required for the program/end-user.

Common Deployment Patterns

As mentioned the Resource Management Application provides a more fine grained decision making as a opposed to the traditional container cluster package management. However, to provide the most flexibility in a production setting not every application may require the need for a detailed RM managed SDR application. Each program/end-user will vary in their actual use case, but controlling the variations deployments are made will simplify the number of maintained deployment methods.

A simple way to describe the types of deployment models is that there is two ways to deploy software applications. The first being SDR based deployments and the second is non-SDR based deployments.

The following table depicts the select common and supported deployment models in a multi-app and multi-SDR environment.

Deployment Tool	Environment	RF/Radio Enabled	Deployment Type
WSPS Resource Management	Production	Yes	SDR Based
Helm Package Manager	Production	Yes	Non-SDR Based
Docker Compose	Development	No	SDR/Non-SDR Based

Ideally from a Production perspective the recommended approach is to only maintain 1 toolset for each Deployment Type. Obviously the RM toolset is primarily driven by gRPC/Protobuf for SDR deployment types which is described in the prior and later sections of this document. The non-SDR based deployment type chosen application deployment is called Helm. Helm provides a generic YAML based approach normally called "Charts". These YAML definitions can be stored along side your application in the repository and utilize third party Helm deployment UI to manage the deployment of non-SDR

based applications. These can also be coupled with kubernetes base UI.

From a Development perspective, normally the use cases and or testing are at a more simplified level that at the system level, so opting for some less resource intensive deployment strategies Docker-Compose

provides a simplified way to enable the software developer to operate the applications allowing the developer(s) to reach a higher level automated unit/integration/system like validation and verfication.

It can also provide a simplified learning curve, where the Docker Compose YAML used to define a deployment, can be converted into the appropriate form for Helm making the transition from a development perspective easier.

Application Integration

The DeviceManager controls deployment, statusing, and teardown of SDR applications within a radio cluster. Applications managed by DeviceManager must meet a set of requirements in order to be integrated with these functions. This document describes the integration requirements as well as some suggestions on preparing your application for integration. This document also describes how to create a 'stack' of applications along with any definitions of persistent services that the applicaiton stack depends on.

App / App Stack Requirements

Dockerization

In order to integrate with the DeviceManager, your application must be Dockerized. A docker image with the application must be available, and the application must run successfully in Docker on the target platform.

Topology

Each application that DeviceManager deploys must be statically configured for a specific radio node. If you want to run multiple related applications distributed across nodes, you will need to work with the DeviceManager team to define the topology so that the correct configuration can be established.

Networking

At present, the DeviceManager runs all docker containers on the host network. This places the following restrictions on your application configuration: - All applications in your application stack must use unique ports - All server applications must bind sockets on all interfaces (`0.0.0.0` for IPv4) - For client and server applications that will be collocated on a single node, the client application can connect to localhost - Client applications hosted on a seperate node from their server must be capable of accepting the hostname of their server via configuration, command line arguments, or environment variables

Logging

In order to ease integration, it is necessary for applications to log all log messages to stdout. Messages that are not logged to stdout will not be accessible.

Status

The DeviceManager (via ClusterResourceManager) can poll applications for status to return to the users. In order to integrate with this status feature, your application must implement the ApplicationService API (see application service). More specifically, it must implement the Status function in that API.

If your application does implement the ApplicationService API, the port used for the endpoint must be configurable via a command line argument, and you must following the requirements under Networking above.

Start / Stop

If your application will be deployed by the DeviceManager as a**persistent service**, the DeviceManager will invoke the "Start" function from the ApplicationService API when an application requiring that persistent service is deployed. When dependent applications are terminated, the DeviceManager will invoke the "Stop" function from the ApplicationService API.

Important Note: Start/Stop will NOT be invoked if your application is deployed as a regular application. Make sure you communicate with the DeviceManager team to determine whether your application will be deployed as a persistent service or as a normal application

Configuration

At present, any configuration file you want to use with your application**must** be included in the Docker image for that container. It is not currently possible to include an external configuration to the application at runtime. The embedded configuration should set autostart to true for an application, such that when the deployment is created the image will be started with the configuration as a command line argument and the configuration will direct the

application to automatically start. There is an exception to this rule for persistent services; however, **make sure you communicate with the DeviceManager team if you think you want your application to be deployed as a persistent service**.

Checklist

The following items must be provided to the DeviceManager team in order to integrate a new application or application stack with the DeviceManager.

- [ ] Application / application stack has been successfully run within Docker in the target environment
- For each application:
  - [ ] Docker image name (repository:tag) provided
  - [ ] Docker image ID (hash) provided
  - [ ] List of all arguments provided to Docker when running the application / application stack in Docker in the target environment
  - [ ] Entrypoint for the application
  - [ ] List of all command line arguments required to run the application
  - [ ] List of all ports used by the application (internal **and** external)
  - [ ] Indicate whether the application requires a GPU
  - [ ] Indicate whether the application requires an SDR
    - [ ] If so, indicate which SDR and / or which RAL config is being used
  - [ ] If the application depends on any persistent service, provide the name of the persistent service and the configuration you wish to apply when your application takes ownership over the service
  - [ ] If the application implements the ApplicationService API, provide the command line argument used to set the port for the ApplicationService API server

Create and Load App Stack Definitions and PrstSvc Definitions

Sections: - Create the directory and files necessary for an App Stack DefinitionStackDefinition - Create the directory and files necessary for a PrstSvcClass Definition PrstSvcDefinition - Load an App Stack and PrstSvcClass DefinitionsLoadDefinitions - Checklist for creating a stack and prstsvc definitions StackDefinitionChecklist - API for using the stack definitions (MissionService.proto API)StackAPI

StackDefinition

Create the directory and files necessary for an App Stack Definition

1. Navigate to $GIT_HOME/device_management/config/ImportDefinitions/ or

2. Create a directory for the stack definition(s) desired to be imported.

- The directory name will be the name of the stack loaded into the database
  - If the container image is a 'managed-app-stub' and used for testing in the resource manager Local Environment, then create the stack definition as follows:
    - ex. $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/**stacks**/ *YourStackDefinitionDirectory*
  - If the container images are for real-world use cases, then create the stack definition as follows:
    - ex. $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/**stacks**/ *YourStackDefinitionDirectory*

3. Example of a finalized directory structure for a Stack.

- ~/**YourDirectoryName**/**stacks**/ *YourStackDefinitionDirectory*
  - App1
    - default-config.json
    - definition.json
    - deployment.json
    - prstsvc-config.json
  - App2
    - default-config.json
    - definition.json
    - deployment.json
  - app-deployment-order.json

4. Create the necessary files for the stack definition(s)

- The following paragraphs contain examples that showcase the most complex use cases for each file required for the stack definition. The available templates and examples for creating files are as follows:

1. For a directory created under $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/dvb-managed-app-stack
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/managed-app-stack1
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/managed-app-stack2
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/ModemSvcDev0StubStack (uses logical resources)
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/ModemSvcDev1StubStack (uses logical resources)
2. For a directory created under $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/dvb-stack
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/modem1-stack
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/modem-dev0-stack (uses logical resources)
    - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/modem-dev1-stack (uses logical resources)

- A stack directory consists of a folder or folders entitled with the stack definition name. The sub directories of each stack definition folder consists of application directories. Each application directory has the following:

    1. **definition.json** - (DO NOT MODIFY THIS FILE NAME) - This file specifies the stack definition and includes key value pairs for 'FileName' of the app k8sdeployment file, the app default config, and any app dependency config files. The AppName should be identical to the name of application directory where the definition.json file lives. Do not add or remove json properties from the definition.json. The app dependencies will be started in the order that they are listed in the AppDependencies array of the definition.json file. If the application is not statusable, then use statusable=false. If the application is statusable, then the stack importer will automatically include the necessary key value pairs in the metadata.labels for the status configuration. Example: $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/ModemSvcDev0StubStack/dechannelizer-dev0-stub/definition.json

```
{
  "Name": "dechannelizer-dev0-stub",    // (required) - this value should match the app directory name
  "K8sDeployment": {"FileName":"deployment.json"},      // (required) - this file should exist in the application dire
  "DefaultConfig": {"FileName":"default-config.json"},  // (required) - this file should exist in the application dire
  "AppDependencies": [        // (optional) - create a curly bracketed definition for each app dependency
      {
          "PrstSvcClass": "RALD-STUB", // (required) - this value should match the name of the PrstSvcClass e.g. */Loc
          "DefaultConfig": {"FileName":"prstsvc-config.json"}, // (required) - this file should exist in the applicati
          "LogicalResource": "dev0",  // (optional) - this value should match the name of a logical resource (see Logi
          "State": "STARTED"  // (required)
      }
  ],
  "StatusConfig": {   // (optional) - include in the app definition only if the app endpoint is statusable.  Loader wi
      "Statusable": true,
      "Port": 30012
  }
}
```

    2. **deployment.json** - This file is the k8s deployment file for the application. This file's name and extension must be specified in the definition.json. Example: $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/ModemSvcDev0StubStack/dechannelizer-dev0-stub/deployment.json

```
{
  "kind": "Deployment",
  "apiVersion": "apps/v1",
  "metadata": {
    "name": "dechannelizer-dev0-stub",
    "creationTimestamp": null,
    "labels": {
    "env": "dechannelizer-dev0-stub_test",
    "msn": "msn1"
    }
  },
  "spec": {
    "replicas": 1,
    "selector": {
    "matchLabels": {
      "env": "dechannelizer-dev0-stub_test",
      "msn": "msn1"
    }
    },
    "template": {
    "metadata": {
      "creationTimestamp": null,
      "labels": {
      "env": "dechannelizer-dev0-stub_test",
      "msn": "msn1"
      }
    },
    "spec": {
      "containers": [
      {
        "name": "dechannelizer-dev0-stub",
        "image": "managed-app-stub:1.1",
        "args": [
          "30012"
        ],
        "resources": {},
        "terminationMessagePath": "/dev/termination-log",
        "terminationMessagePolicy": "File",
        "imagePullPolicy": "IfNotPresent"
      }
      ],
      "restartPolicy": "Always",
      "terminationGracePeriodSeconds": 30,
      "dnsPolicy": "ClusterFirst",
      "nodeSelector": {
      "kubernetes.io/hostname": "kube-worker1.local"
      },
      "hostNetwork": true,
      "securityContext": {},
      "schedulerName": "default-scheduler"
    }
    },
    "strategy": {
    "type": "RollingUpdate",
    "rollingUpdate": {
      "maxUnavailable": "25%",
      "maxSurge": "25%"
    }
    },
    "revisionHistoryLimit": 10,
    "progressDeadlineSeconds": 600
  },
  "status": {}
}
```


3. **default-config.json** - This file is the application default config. This file's name and extension must be specified in the definition.json. If no configuration is necessary, then use '{}' as the contents of the file. Since most kubernetes app json files point to an internal config, the default-

config.json is used for documentation purposes for the operator. It represents the default configuration that the application json uses to start the application container. Example: $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/modem-dev0-stack/dechannelizer-deployment-dev0/default-config.json

```
{
  "autoStart": true,
  "DeviceID": 0,
  "WidebandCenterFrequencyHz": 1500000000,
  "WBSampleRateHz": 62500000,
  "NBSampleRateHz": 250000,
  "NumTuners": 100,
  "FrequencyShiftHz": 0,
  "MaxBlocksReadAheadOfWrite": 3,
  "WBBlockSize": 500000,
  "NBBlockSize": 2000,
  "picNumber": 0,
  "MinScale": 10000.0,
  "TunerBank":
  {
      "Name": "ModemTx",
      "ChannelBandwidthHz": 62500,
      "RRCRollOff": 0.35,
      "RRCOverSampleRate": 4,
      "RRCTaps": 16
  }
}
```

4. **prstsvc-config.json** - This file is the appDependency (prstSvc) override config. This config will override the default config of the prstSvc. This file's name and extension must be specified in the definition.json. If no configuration is necessary, then use '{}' as the contents of the file. Example: $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks/modem1-stack/EttusDevice.json

```
{
  "autoStart": true,
  "devicePlugin":
  {
      "name": "EttusDevice",
      "config":
      {
          "ClockRate": 125000000,
          "UseExternalReference": true,
          "UseExternalTrigger": true,
          "DeviceAddress": "192.168.20.2",
          "RequireNtp":true,
          "transmitterConfig":
          {
              "MaxRefLockAttempts": 100,
              "ChannelConfig":
              {
                  "Channel": 0,
                  "SampleRate": 62500000,
                  "TxFrequencyHz": 1500000000,
                  "LOOffsetHz": 30000000,
                  "GaindB": 30.0
              },
              "apiProviderPlugin":
              {
                  "name": "IcemTransmitterAPIProvider",
                  "config":
                  {
                    "Card":0,
                    "DmaSize": 49725000,
                    "NumSamples": 49725,
                    "SampleRate": 62500000,
                    "NumReadAheadBuffers": 5
                  }
              }
          },

          "receiverConfig":
          {
              "BufferSize": 187500,
              "Channel": 0,
              "SampleRate": 62500000,
              "RxFrequencyHz": 1500000000,
              "LOOffsetHz": 25000000,
              "GaindB": 30.0,
              "UseFractionalTuning": true,
              "apiProviderPlugin":
              {
                  "name": "IcemReceiverAPIProvider",
                  "config":
                  {
                      "cardNum": 1,
                      "dmasize": 250000000
                  }
              }
          }
      }
  }
}
```

- If the application deployment order matters when the stack is deployed, then in perform the following steps
  1. In the directory of *YourStackDefinitionDirectory* include a file named exactly "app-deployment-order.json"
  2. Update the file of the json array with the exact names of the application directories in order of deployment (The AppName in the definition.json file should match the application directory name). Example: $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks/dvb-managed-app-stack/app-deployment-order.json `json [ "dvb-controller-service", "dvb-return-channel-service", "dvb-forward-channel-service", "dvb-mpeg-`

```
deserialize-service", "dvb-snmp" ]
```

PrstSvcDefinition

Create the directory and files necessary for a PrstSvcClass Definition

1. Navigate to $GIT_HOME/device_management/config/ImportDefinitions/

2. Create a directory for the prstSvcClass definition(s) desired to be imported.

- The directory name will be the name of the prstSvcClass loaded into the database
  - If the container image is a 'managed-app-stub' and used for testing in the resource manager Local Environment, then create the stack definition as follows:
    - ex. $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/**prstSvcClasses**/ *YourPrstSvcClassDefinitionDirectory*
  - If the container images are for real-world use cases, then create the stack definition as follows:
    - ex. $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/**prstSvcClasses**/ *YourPrstSvcClassDefinitionDirectory*

3. Example of a finalized directory structure for PrstSvcs.

- ~/**YourDirectoryName**/**prstSvcClasses**/
  - *YourPrstSvcClassDefinitionDirectory* (ex. PRSTSVC1, RALD-STUB)
    - default-config.json
    - definition.json
    - deployment.json
  - *YourPrstSvcClassDefinitionDirectory* (ex. PRSTSVC2)
    - default-config.json
    - definition.json
    - deployment.json

4. Create the necessary files for the prstSvcClass definition(s)

- The following paragraphs contain examples that showcase the most complex use cases for each file required for the stack definition. The available templates and examples for creating files are as follows:
  1. For a directory created under $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/prstSvcClasses
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/prstSvcClasses/PRSTSVC1
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/prstSvcClasses/PRSTSVC2
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/prstSvcClasses/RAL
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/prstSvcClasses/RALD-STUB
  2. For a directory created under $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RAL
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RALD-MULTI
     - Navigate to $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RALD-SINGLE

- *YourPrstSvcClassDefinitionDirectory* is a directory that consist of files that define the prstSvcClass. The directory consists of the following:

  1. **definition.json** - (DO NOT MODIFY THIS FILE NAME) - this file specifies the prstSvcClass definition and includes key value pairs for 'FileName' of the k8sdeployment file, the default config, and status config information. Do not add or remove json properties from the definition.json. The definition.json file for prstSvcClass definitions does not include a field for 'AppDependencies'. If the persistent service is not statusable, then use statusable=false. If the persistent service is statusable, then the stack importer will automatically include the necessary key value pairs in the metadata.labels for the status configuration. Example: $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RALD-MULTI/definition.json

```
{
  "Name": "RALD-MULTI",   // (required) - this value should match the prstSvcClass directory name
  "K8sDeployment": {"FileName":"deployment.json"},      // (required) - this file should exist in the application dire
  "DefaultConfig": {"FileName":"EttusMultiDeviceConfigN310.json"},  // (required) - this file should exist in the appli
  "LogicalResources": [     // (optional) - list the logical resource name in square bracketed definition
      "dev0",
      "dev1"
  ],
  "StatusConfig": {     // (optional) - include in the app definition only if the app endpoint is statusable.  Loader
      "Statusable": true,
      "Port": 30004
  }
}
```

2. **deployment.json** - This file is the k8s deployment file for the prstSvc. This file's name and extension must be specified in the definition.json. Example: $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RALD-MULTI/deployment.json

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "creationTimestamp": null,
    "labels": {
      "env": "rald_test"
    },
    "name": "modem-rald-multi"
  },
  "spec": {
    "progressDeadlineSeconds": 600,
    "replicas": 1,
    "revisionHistoryLimit": 10,
    "selector": {
      "matchLabels": {
        "env": "rald_test"
      }
    },
    "strategy": {
      "rollingUpdate": {
        "maxSurge": "25%",
        "maxUnavailable": "25%"
      },
      "type": "RollingUpdate"
    },
    "template": {
      "metadata": {
        "creationTimestamp": null,
        "labels": {
          "env": "rald_test"
        }
      },
      "spec": {
        "containers": [
          {
            "name": "rald",
            "command": [
              "/opt/ws/bin/rald"
            ],
            "args": [
              "--debug",
              "--config",
              "/opt/ws/config/ModemService/ralConfig.json",
              "--grpcport",
              "30004"
            ],
            "image": "wsps-docker-us-only.lmco-art.us.lmco.com:8443/uhd_3_14/ral/rald/app:uhd_3_14",
            "imagePullPolicy": "IfNotPresent",
```

```
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "terminationMessagePolicy": "File"
          }
        ],
        "dnsPolicy": "ClusterFirst",
        "hostIPC": true,
        "hostNetwork": true,
        "nodeSelector": {
          "kubernetes.io/hostname": "wsrad04.webdev.local"
        },
        "restartPolicy": "Always",
        "schedulerName": "default-scheduler",
        "securityContext": {},
        "terminationGracePeriodSeconds": 30,
        "imagePullSecrets": [
          {
            "name": "nexus-cred"
          }
        ]
      }
    }
  },
  "status": {}
}
```

3. **default-config.json** - This file is the prstSvc default config. This file's name and extension must be specified in the definition.json. If no configuration is necessary, then use '{}' as the contents of the file.

For RALD-MULTI: the kubernetes prstsvc json will have an internal config of "EmptyDeviceConfig.json", which is {"autoStart:false}. The default-config.json for RALD-MULTI that is loaded into redis is named "blankReferenceConfig.json" and the content is "{}". This setup allows the application's version of the RALD-MULTI config to cleanly override a blank reference config.

Example: $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/prstSvcClasses/RALD-MULTI/EttusMultiDeviceConfigN310.json

```
{
  "autoStart": false,
  "devicePlugin": {
      "name": "EttusMultiDevice",
      "config": {
          "ClockRate": 125000000,
          "UseExternalReference": false,
          "UseExternalTrigger": false,
          "DeviceAddress": "192.168.20.2",
          "transmitterConfig": {
              "MaxRefLockAttempts": 100,
              "NsToTransmit": 1000000,
              "SampleRate": 62500000,
              "ChannelConfig": [
                  {
                      "Channel": 0,
                      "ChannelName": "myTxChannel",
                      "DeviceName": "A:0",
                      "TxFrequencyHz": 1500000000,
                      "LOOffsetHz": 30000000,
                      "GaindB": 31,
                      "apiProviderPlugin": {
                          "name": "NetworkTransmitterAPIProvider",
                          "config":
                          {
                            "RingBufferSize": 62500000,
                            "Protocol": "TCP",
                            "Endpoint": "127.0.0.1:9999"
                          }
                      }
                  }
              ]
          },
          "receiverConfig": {
              "SampleRate": 62500000,
              "nsToReceive": 1000000,
              "ChannelConfig": [
                  {
                      "Channel": 0,
                      "ChannelName": "myRxChannel",
                      "DeviceName": "A:0",
                      "RxFrequencyHz": 1500000000,
                      "LOOffsetHz": 25000000,
                      "GaindB": 31,
                      "UseFractionalTuning": true,
                      "apiProviderPlugin": {
                          "name": "NetworkReceiverAPIProvider",
                          "config":
                          {
                            "Protocol": "TCP",
                            "Endpoint": "127.0.0.1:9999"
                          }
                      }
                  }
              ]
          }
      }
  }
}
```


Appsettings

Notes about the appsettings.json and appsettings.WEBDEV.json for the DefinitionsImportDirectory field. 1. For a stack directory created under $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/stacks - The DefinitionsImportDirectory in the appsettings.json should be: "../../../../device_management/config/ImportDefinitions/LocalEnvironment" Example: $GIT_HOME/device_management/DeviceManager/src/DeviceManager.Main/appsettings.json `json { ... "DefinitionsImportDirectory":`

`"../../../../device_management/config/ImportDefinitions/LocalEnvironment", ... }` 2. For a stack directory created under $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/stacks - The DefinitionsImportDirectory in the appsettings.WEBDEV.json should be: "../../../../device_management/config/ImportDefinitions/NonLocalEnvironment" Example: $GIT_HOME/device_management/DeviceManager/src/DeviceManager.Main/appsettings.WEBDEV.json `json { ... "DefinitionsImportDirectory": "../../../../device_management/config/ImportDefinitions/NonLocalEnvironment", ... }` 3. For a stack directory created under /**YourDirectoryName**/ - The DefinitionsImportDirectory in the appsettings.json should be: "../../../../**YourDirectoryName**" - The path must have the correct number of '../' patterns such that the stack directory location is found by the appsettings configuration manager

LoadDefinitions

Load an App Stack and PrstSvcClass Definitions - Load upon running DM: run DM using 'dotnet run -s' - Run DM without loading: run DM using 'dotnet run'

StackDefinitionChecklist

This is a quick checklist for how to create stack and prstsvc definitions - [ ] Create and name a parent directory for the*stacks* definition(s) and *prstSvcClasses* definition(s) or use an existing directory - See Step 1 and 2 of StackDefinition and PrstSvcDefinition for more information - /**YourDirectoryName**/ - Or use one of the existing directories: - $GIT_HOME/device_management/config/ImportDefinitions/LocalEnvironment/ - $GIT_HOME/device_management/config/ImportDefinitions/NonLocalEnvironment/ - [ ] Update the appsettings*.json 'DefinitionsImportDirectory' the path to /**YourDirectoryName**/ - See Appsettings for more information - [ ] Create the main directory of stack definitions ex. /**YourDirectoryName**/**stacks**/ - [ ] Create the main prstSvcClass directory ex. /**YourDirectoryName**/**prstSvcClasses**/ - Create a stack definition - See Step 3 of StackDefinition for complete instructions, templates, examples, and explanations - In the directory */**stacks**/ - [ ] Create the application directory (repeat for each application) - Create the following application files using the information gathered from Checklist - [ ] definition.json (required) - [ ] deployment.json (required) - [ ] default-config.json (optional) - [ ] prstsvc-config.json (optional) - [ ] If the application deployment order matters, then create the "app-deployment-order.json" file under the stack directory - Create a prstsvc definition - See Step 3 of PrstSvcDefinition for complete instructions, templates, examples, and explanations - In the directory */**prstSvcClasses**/ - [ ] Create the prstSvcClass directory (repeat for each prstsvc) - Create the following application files using the information gathered from Checklist - [ ] definition.json (required) - [ ] deployment.json (required) - [ ] default-config.json (optional) - [ ] prstsvc-config.json (optional)

StackAPI

API endpoints for using the loaded stack definitions. These endpoints are defined in MissionService.proto in the api_services repo. The enpoints can be accessed via the hmi in engineering-sw-tools repo, or the resource manager gui in the resource_manager_gui repo.
The endpoints related to working with stack definitions are as follows: 1. createStackInstance() 2. listStacknames() 3. getStackInstanceStatuses() 4. listNodenames() - used to find out which nodes a stack can be deployed on.