ERDK PREFERRED PROGRAM

RBUS – How it's all connected

Charles Moreman, Comcast
Karunakaran Amirthalingam, Comcast

WEBINAR SPONSORED BY

5VVideo.
arcadyan

//consult.red
DTVKit
GCS RECRUITMENT SPECIALISTS

HUMAX NETWORKS
Infosys
irdeto

Sagemcom
SERCOMM
SKYWORTH
TATA ELXSI
TV2Z

What is RBUS?

- RBUS is a generic inter-process communication system.
- It is designed for small memory footprint devices.
- It enables fast message passing between software components.
- Software components use RBUS to access or manage services and features in other software components.
- RBUS is only for internal messages within a device. RBUS is not used for external messages. Note: Software components can support messages over various external connections and can pass these messages to RBUS.

RBUS Features

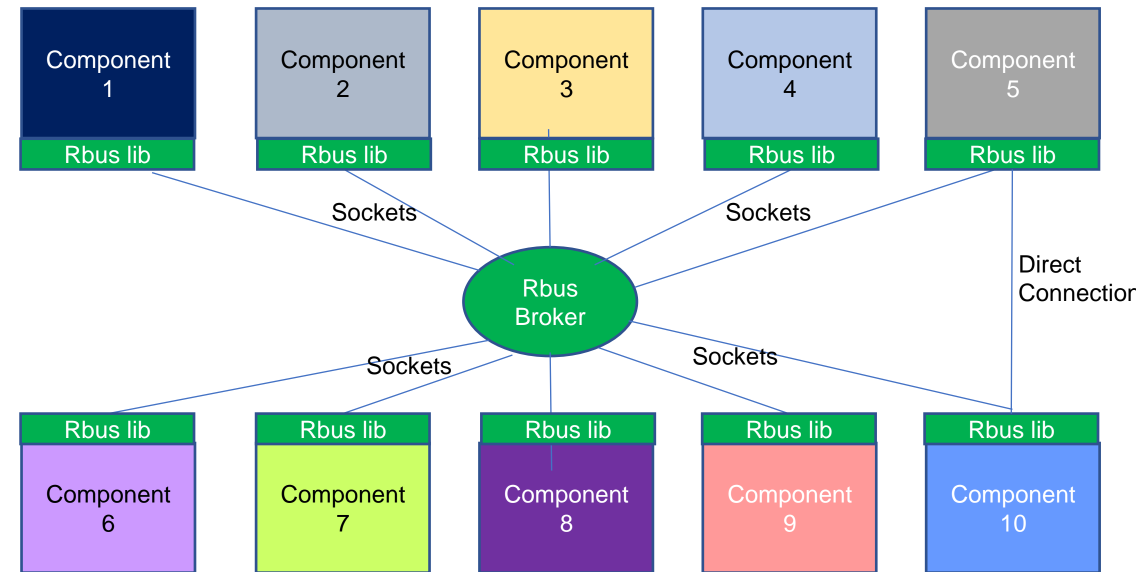
RBUS enables software components to communicate with each other to:

- Set a value in another component
- Read a value in another component
- Subscribe to a value change or event in another component
- Subscribe to a property value to receive regularly occurring updates.
- Get notifications in response to a previous subscription
- Invoke a Command located in another component
- And other things. See more details in the following slides.

RBUS Overview – How does it work?

- Software components use the **RBUS APIs**.
- These APIs are provided by a library linked to each component.
- The RBUS **Broker** (Rbus daemon) routes messages to the correct destination.
- The broker operation is transparent to the components using the RBUS APIs.
- RBUS also supports optional **Direct Connections** for high volume or latency sensitive messages.

  Note: Use Direct Connections only when needed. A Direct Connection between all components requires more resources compared to brokered connections.



RBUS APIs – Consumers and Providers

- A software component may “consume” services that are “provided” by other components. That same component may “provide” services to other components
- Consumer API examples:
  - Get, set property values. Add and remove rows in tables.
  - Subscribe to events.
  - Invoke remote methods.
  - Discover other software components
- Provider API examples:
  - Register properties and implement their get and set operations.
  - Register handlers for add / remove operations for table rows.
  - Register and publish notification messages.
  - Register and implement commands (methods) which can be invoked from other components

RBUS Overview – A few examples:

1. A telemetry component can use RBUS to collect data from other components.
2. The “WAN Manager” Component can use RBUS invoke a WAN test command in another component
3. A software component can use RBUS to “subscribe” to an event notification when a client device joins WiFi
4. The USP Protocol Adapter (PA) software component allows an external USP controller(s) to manage a device.
   - An external USP controller sends a message to set the WIFI password,
   - The USP PA receives this message and passes it to RBUS.
   - RBUS routes the message to the OneWiFi software component.
   - This component sets the new password value and returns a response back to the USP PA.
   - The USP PA returns the response to the external USP controller.

Let’s look at the details.

API definition files

- Rbus Value (rbus_value.h)
  - Variant data type that can hold, well, a value (for basic and extended data types).
- Rbus Property (rbus_property.h)
  - A name and value pair. Could be a single or array of properties.
- Rbus Object (rbus_object.h)
  - A named collection of properties.
- Rbus (rbus.h)
  - The primary interface that provides APIs for consumers and providers (and uses other interfaces as needed).

API Overview - rbus_open( )

- Software components must connect to the bus before sending messages

```
rbusError_t rbus_open(rbusHandle_t* handle, char *componentName);
```

- When rbus_open() is called, the system creates a connection between that software component and the rbus broker
- Two different type sockets are supported
  - When a component is in the same CPU (or virtual container), the connection uses a Unix domain socket.
  - When a component is in a different CPU, the connection uses a TCP socket.
- Special Case Note: When multiple software components within the same software process call rbus_open() a new rbusHandle is created but the physical socket connection is shared.

API Overview - rbus_openDirect( )

- Software components can request a direction connection to another component.

```
rbusError_t rbus_openDirect(rbusHandle_t handle, rbusHandle_t* myDirectHandle, char const* propertyName)
```

- When rbus_openDirect() is called, the system creates a direct socket connection between that software component and another component that provides this property or method.
- Direct connections should only be used in special cases where high traffic volumes or latency sensitive messages are exchanged.
- When a component is in the same CPU (or virtual container), the connection uses a Unix domain socket.
  - When a component is in a different CPU, the connection uses a TCP socket.
- Special Case Note: Use Direction Connections only when needed. Direct connections between all components would use more resources than brokered connections.

API Overview – Register names to the system

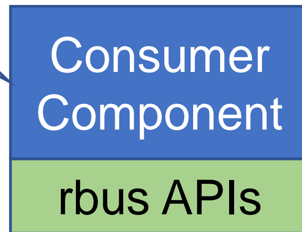
- A component that needs to provide services to the system must "Register" a **unique name** for each property or service. This also registers **callbacks** that are invoked when a message is received at each name.
- These names are used as the destination name to route messages across the system.
- When the rbus broker receives a message, it routes the message to the component that has registered this name.

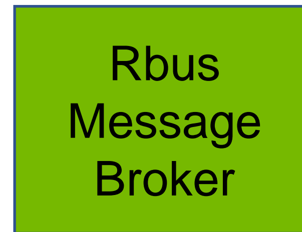
```
rbusError_t rbus_regDataElements(rbusHandle_t handle, int numDataElements, rbusDataElement_t *elements);
```

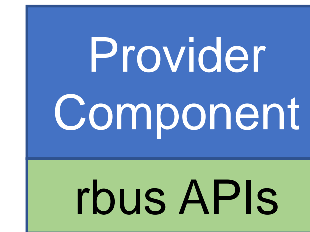
API Overview – rbus_set() and rbus_get()

RBUS supports reading and writing a property value in another component

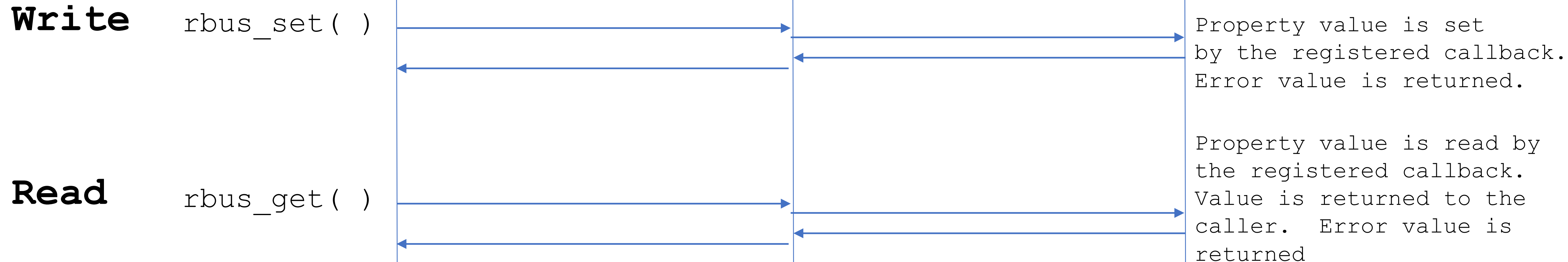
This component needs to read or write a property value







This component provides the unique property name to the system




```
rbusError_t rbus_set(rbusHandle_t handle, char const* name, rbusValue_t value, rbusSetOptions_t* opts);
rbusError_t rbus_get(rbusHandle_t handle, char const* name, rbusValue_t* value);
```

API Overview – Simple example to set the value of a property

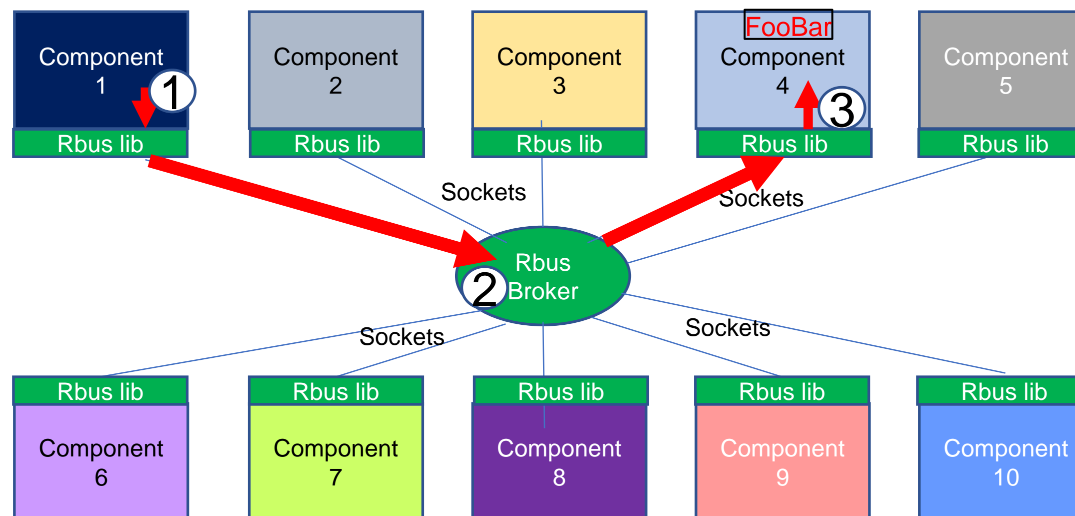
Example of how rbus works:

- Component 4 previously registered a property with the unique name "FooBar". This property supports writing/reading an integer value. This component is known as the "provider" of the property named "FooBar".

① Component 1 calls an API to set the value of FooBar = 2.

② The rbus broker receives the message from Component 1. It does an internal lookup to see which outgoing socket to use for destination "FooBar". It forwards the message to Component 4.

③ A calback is invoked in Component 4 that handles writes to FooBar. It sets the value to 2. rbusError (success) is returned to Component 1 (not shown)



APIs overview (contd.)

- Registration

```
typedef enum {
    RBUS_ELEMENT_TYPE_PROPERTY  = 1,    /**< Property Element. Sample names: x.y, p.q.{i}.r, aaa, etc.
                                             Can also be monitored and event notifications be obtained
                                             in the form of events                                    */
    RBUS_ELEMENT_TYPE_TABLE,            /**< Table (e.g. multi-instance object).
                                             Sample names: a.b.{i}, a.b.{i}.x.y.{i}                   */
    RBUS_ELEMENT_TYPE_EVENT,            /**< (Exclusive) Event Element. Sample names: a.b.c!, zzzz!   */
    RBUS_ELEMENT_TYPE_METHOD            /**< Method Element. Sample names: m.n.o(), dddddd()          */
} rbusElementType_t;
```

```
typedef struct rbusCallbackTable_t{
    rbusGetHandler_t                getHandler;
    rbusSetHandler_t                setHandler;
    rbusTableAddRowHandler_t        tableAddRowHandler;
    rbusTableRemoveRowHandler_t     tableRemoveRowHandler;
    rbusEventSubHandler_t           eventSubHandler;
    rbusMethodHandler_t             methodHandler;
} rbusCallbackTable_t;
```

```
typedef struct{
    char*                       name;
    rbusElementType_t           type;
    rbusCallbackTable_t         cbTable;
}rbusDataElement_t;
```

```
rbusError_t rbus_regDataElements(rbusHandle_t handle, int numDataElements, rbusDataElement_t *elements);
```

APIs overview (contd.)

- Callback Handlers (quick look..)

```
typedef rbusError_t (*rbusGetHandler_t)(rbusHandle_t handle, rbusProperty_t property,
                                                             rbusGetHandlerOptions_t* options);

typedef rbusError_t (*rbusSetHandler_t)(rbusHandle_t handle, rbusProperty_t property,
                                                             rbusSetHandlerOptions_t* options);

typedef rbusError_t (*rbusTableAddRowHandler_t)(rbusHandle_t handle, char const* tableName,
                                                     char const* aliasName, uint32_t* instNum);

typedef rbusError_t (*rbusTableRemoveRowHandler_t)(rbusHandle_t handle, char const* rowName);

typedef rbusError_t (* rbusEventSubHandler_t)(rbusHandle_t handle, rbusEventSubAction_t action,
                     char const* eventName, rbusFilter_t* filter, int interval,  bool* autoPublish);

typedef rbusError_t (*rbusMethodHandler_t)(rbusHandle_t handle, char const* methodName,
                     rbusObject_t inParams, rbusObject_t outParams, rbusMethodAsyncHandle_t asyncHandle);
```

APIs overview (contd.)

- Discovery

```
rbusError_t rbus_discoverComponentName(rbusHandle_t handle,
                                        int numElements,
                                        char** elementNames,
                                        int *numComponents,
                                        char ***componentName);


rbusError_t rbus_discoverComponentDataElements(rbusHandle_t handle,
                                        char* name,
                                        bool nextLevel,
                                        int *numElements,
                                        char*** elementNames);
```

APIs overview (contd.)

- Get

```
rbusError_t rbus_get(rbusHandle_t handle, char const* name, rbusValue_t* value);

rbusError_t rbus_getExt(rbusHandle_t handle, int paramCount, char const** paramNames,
                                                            int *numProps, rbusProperty_t* properties);

rbusError_t rbus_getInt(rbusHandle_t handle, char const* paramName, int* paramVal);

rbusError_t rbus_getUint(rbusHandle_t handle, char const* paramName, unsigned int* paramVal);

rbusError_t rbus_getStr(rbusHandle_t handle, char const* paramName, char** paramVal);
```

APIs overview (contd.)

- Set

```
rbusError_t rbus_set(rbusHandle_t handle, char const* name, rbusValue_t value,
                                                                     rbusSetOptions_t* opts);

rbusError_t rbus_setMulti(rbusHandle_t handle, int numProps, rbusProperty_t properties,
                                                                     rbusSetOptions_t* opts);

rbusError_t rbus_setInt(rbusHandle_t handle, char const* paramName, int paramVal);

rbusError_t rbus_setUInt(rbusHandle_t handle, char const* paramName, unsigned int paramVal);

rbusError_t rbus_setStr(rbusHandle_t handle, char const* paramName, char const* paramVal);
```

APIs overview (contd.)

- Table operations

```
rbusError_t rbusTable_addRow(rbusHandle_t handle, char const* tableName,
                             char const* aliasName, uint32_t* instNum);


rbusError_t rbusTable_removeRow(rbusHandle_t handle, char const* rowName);
```

APIs overview (contd.)

- Events / Notifications

```
// Rbus filter is a more sophisticated filter that
// allows both logical and relational operations
// to be performed on the value to filter out
// certain values, before the event is delivered.
// Refer rbus_filter.h for more info.
typedef struct _rbusFilter* rbusFilter_t;

typedef struct _rbusEventSubscription{
    char const*          eventName;
    rbusFilter_t*        filter;
    int32_t              interval;
    uint32_t             duration;
    bool                 publishOnSubscribe;
    void*                userData;
} rbusEventSubscription_t;
```

```
typedef enum{
    RBUS_EVENT_OBJECT_CREATED,
    RBUS_EVENT_OBJECT_DELETED,
    RBUS_EVENT_VALUE_CHANGED,
    RBUS_EVENT_GENERAL,
    RBUS_EVENT_INITIAL_VALUE,
    RBUS_EVENT_INTERVAL,
    RBUS_EVENT_DURATION_COMPLETE
} rbusEventType_t;

typedef struct{
    char const*          name;
    rbusEventType_t      type;
    rbusObject_t         data;
}rbusEvent_t;
```

APIs overview (contd.)

- Events / Notifications (contd.)

```
typedef void (*rbusEventHandler_t)(rbusHandle_t handle, rbusEvent_t const* eventData,
                                                    rbusEventSubscription_t*    subscription);

rbusError_t  rbusEvent_Subscribe(rbusHandle_t handle, char const* eventName,
                                                    rbusEventHandler_t  handler, void* userData);

rbusError_t rbusEvent_SubscribeEx(rbusHandle_t handle, rbusEventSubscription_t* subscription,
                                                                            int numSubscriptions);

rbusError_t  rbusEvent_Publish(rbusHandle_t handle, rbusEvent_t* eventData);

rbusError_t rbusEvent_Unsubscribe(rbusHandle_t handle, char const* eventName);

rbusError_t rbusEvent_UnsubscribeEx(rbusHandle_t handle, rbusEventSubscription_t* subscription,
                                                                            int numSubscriptions);
```

APIs overview (contd.)

- Methods

Consumer Side:

```
rbusError_t rbusMethod_Invoke(rbusHandle_t handle, char const* methodName,
                                                    rbusObject_t inParams, rbusObject_t* outParams);

typedef void (*rbusMethodAsyncRespHandler_t)(rbusHandle_t handle, char const* methodName,
                                                            rbusError_t error, rbusObject_t params);

rbusError_t rbusMethod_InvokeAsync(rbusHandle_t handle, char const* methodName,
                          rbusObject_t inParams, rbusMethodAsyncRespHandler_t callback, int timeout);
```

Provider Side:

```
typedef rbusError_t (*rbusMethodHandler_t)(rbusHandle_t handle, char const* methodName,
                 rbusObject_t inParams, rbusObject_t outParams, rbusMethodAsyncHandle_t asyncHandle);

rbusError_t rbusMethod_SendAsyncResponse(rbusMethodAsyncHandle_t asyncHandle,
                                                     rbusError_t error, rbusObject_t outParams);
```

APIs overview (contd.)

- Callback Handlers (revisit)

```
typedef rbusError_t (*rbusGetHandler_t)(rbusHandle_t handle, rbusProperty_t property,
                                                             rbusGetHandlerOptions_t* options);

typedef rbusError_t (*rbusSetHandler_t)(rbusHandle_t handle, rbusProperty_t property,
                                                             rbusSetHandlerOptions_t* options);

typedef rbusError_t (*rbusTableAddRowHandler_t)(rbusHandle_t handle, char const* tableName,
                                                     char const* aliasName, uint32_t* instNum);

typedef rbusError_t (*rbusTableRemoveRowHandler_t)(rbusHandle_t handle, char const* rowName);

typedef rbusError_t (* rbusEventSubHandler_t)(rbusHandle_t handle, rbusEventSubAction_t action,
                 char const* eventName, rbusFilter_t* filter, int interval,  bool* autoPublish);

typedef rbusError_t (*rbusMethodHandler_t)(rbusHandle_t handle, char const* methodName,
                 rbusObject_t inParams, rbusObject_t outParams, rbusMethodAsyncHandle_t asyncHandle);
```

API Overview - rbus_close ( )

- Software components close the rbus connection if no longer needed

```
rbusError_t rbus_close(rbusHandle_t handle);
```

- When rbus_close() is called, the system removes the connection between that software component and the rbus broker.

RBUS – How it's all connected

Questions?

Thank You