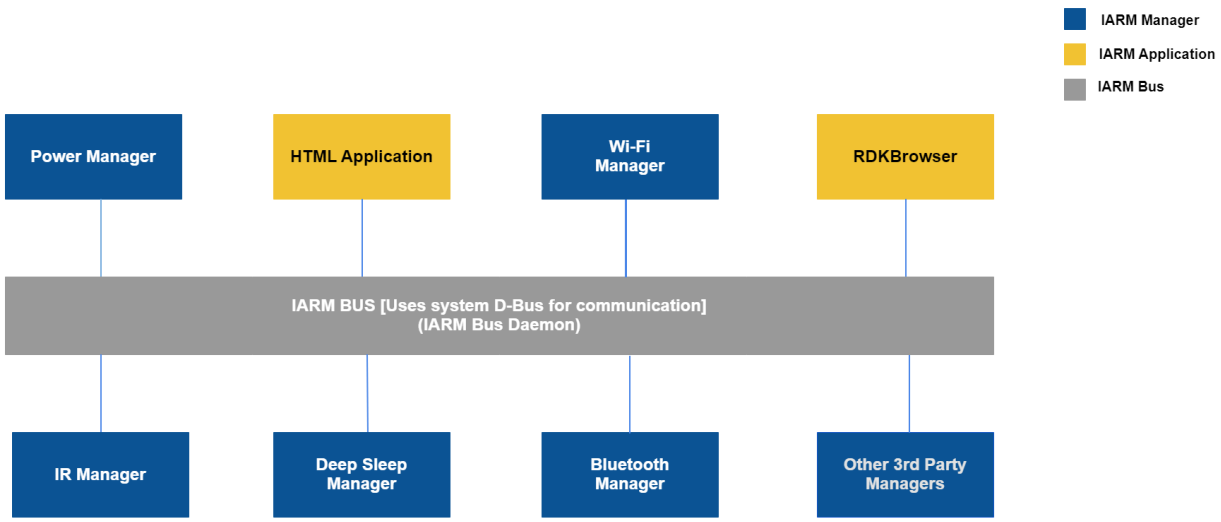
IARM Bus

Overview

IARM-Bus is a platform agnostic Inter-process communication (IPC) interface. It allows applications to communicate with each other by sending ***Events*** or invoking ***Remote Procedure Calls***. The common programming APIs offered by the RDK IARM-Bus interface is independent of the operating system or the underlying IPC mechanism.

Two applications connected to a same instance of IARM-Bus are able to exchange events or RPC calls. On a typical system, only one instance of IARM-Bus instance is needed. If desired, it is possible to have multiple IARM-Bus instances. However, applications connected to different buses will not be able to communicate with each other.



Bus Daemon

Each ***Bus Instance*** is managed by a single ***Bus Daemon Process***. The Bus Daemon oversees the activities on the bus and manage common resources competed by all connected applications. Applications requesting a shared resource must firstly acquire from the Bus Daemon. Once it is done with the resource, it must release the ownership back to Bus Daemon so that other applications can use the resource. The IARM Bus Daemon offers standard APIs to request and release ownership of a resource. In Essence, a Bus Daemon itself is a regular applications connected to the IARM-Bus with special privileges. It is the only entity that has knowledge of all applications connected to the bus, and has the authority of granting or denying resources.

Well-Known Name

Each application that connected to the bus must identify itself with a unique name. This name is considered well-known in that it is used by other applications to send events or invoke RPC calls published by the application. The well-known name thus is considered part of the public API interface published by the application. The Bus daemon's well-known name is **"Daemon".**

An application's well-known name is only visible to the bus that it connects to. When concatenated with the IARM-bus name com.comcast.rdk.iarm.bus, it becomes the application's universally unique identifier.

Programming Guidelines

As explained above, IARM-Bus offers two basic functionalities:

- Send Events to application.
- Invoke application's RPC methods.

> ⚠ When developing IARM APIs, it is strongly recommended developers adopt the naming conventions suggested in this guideline.

An IARM Application that runs as a linux daemon process is considered a **Manager Component**. The IARM-Bus ***Daemon*** is a special Manager component that belongs to the IARM core. Such manager components normally register Events and RPC methods for other applications to use.

> ⚠ Under iarm/template, a template implementation of a generic Manager Component is provided. Developer can make a copy of the templates and modify from it to suit their application needs. The template conforms to the guidelines explained in this section.

The directory under IARM should use this structure:

Directory	Files
$iarm	IARM Core, IARM Bus Daemon, and all IARM Manager Components
core	Core implementation of IARM framework. It is built into a shared library for all IARM applications to use.
core /include	Public headers exposed by IARM core. IARM application should only reference header files located here to use IARM-Bus APIs.
<mgr>	Manager's internal implementation. Header files listed here should NOT be referenced by any external components outside the <mgr> directory.
<mgr> /include	Public headers exposed by the Manager. IARM applications uses the Manager's events or RPC methods should only reference header files located here.
template	A Sample implementation of a IARM Manager Component. This provides a good starting point for developers to create their manager components. The developer can make a copy of the template and modify to suit their application needs.

The IARM-Bus Library is ./core/libIARMBus.so. All IARM Applications must link to this library.
In this document we use the implementation of IR Manager as our example. The source code of IR Manager is available at iarm/ir/.
IR Manager is an application that publishes Remote Key events to other applications.

- IR manager sends these IR events to other applications.
  - IARM_BUS_IRMGR_EVENT_IRKEY
- The Event Data contains Key Code and Key Type of the pressed IR key.
  - int keyType;
  - int keyCode;
- IR manager publishes two RPC Methods:
  - SetRepeatKeyInterval
  - GetRepeatKeyInterval
- Other application can invoke these methods to determine how often a repeat key is sent when the IR key is held own by the user.

We will use IR Manager's implementation to illustrate how to use IARM-Bus APIs to publish, send, receive and handle Events, as well as publish and invoke RPC Methods.
This document explains the usage of IARM-Bus APIs by example. For a full API specification please refer to IARM's public header files or doxygen document.

How to Create, Send, and Receive Events

Declare Well-Known Name in Header File

Since all the events published by the application are tied to its well-known name, the application must make its well-known name available to other applications that wish to use its service. The well-known name can be thought of as the Namespace within which the application's events and RPC methods live. This name must be declared in the application's public header file. In our example, this file is irMgr.h

```
#define IARM_BUS_IRMGR_NAME "IRMgr"
Naming Convention: IARM_BUS_<Manager>_NAME "Manager"
```

When the application initialize with IARM-Bus, it provides its well-known name:

```
IARM_Bus_Init(IARM_BUS_IRMGR_NAME);
IARM_Bus_Connect();
```

Applications should use the defined macro of the owner's well-known name (IARM_BUS_IRMGR_NAME in this example) when sending or listening for events.

Declare Event and Event Data in Header File

Considering the application's well-name as a namespace name, an Event ID is used to uniquely identify the Event within the namespace. The Event IDs must be declared in the application's public header file, using C Enumerations and must start with value **0**.

```
typedef enum _EventId_t {
     IARM_BUS_IRMGR_EVENT_IRKEY = 0,
     IARM_BUS_IRMGR_EVENT_MAX
 }
```

Naming Convention: IARM_BUS_<Manager>EVENT<Event Name>

_**EVENT_MAX** must have the value of (Number Of Events Published + 1). This MAX value is used when registering all events to the IARM –Bus.
An event may or may not have Event Data. If an event has event data, the data types must be declared so that the listeners of the events can process the data properly. The data structure should a union of event data for all events that the application would send. For example, IR Manager's IR Event Data contains a **keyType** and **KeyCode** field.

```
typedef struct _IRMgr_EventData_t {
    union {
       struct _IRKEY_DATA{
           int keyType;
            int keyCode;
       } irkey, fpkey;
   } data;
}IARM_Bus_IRMgr_EventData_t;
```

Naming Convention: IARM_Bus_<Manager>_EventData_t

> ⚠ The event data structure ***cannot*** have pointers. The **sizeof()** operator applied on the _EventData_t must equal to the actual memory allocated. Internally an equivalent of **memcpy()** is used to dispatch event data to its target. If a pointer is used in the event data, the pointer, not the content it points to, is sent to the destination.

Register Event to IARM-Bus

An Event must be Registered with IARM-Bus first before it can be sent out or listened to. The application that registers the event is considered the **Owner** o f the event. During initialization, the owner can register all the events it will send to the bus.
Now that both the event and event data are declared, the application can register all its events to the IARM-Bus. Note how _EVENT_MAX constant is used here to register all events in a single call.

```
IARM_Bus_RegisterEvent(IARM_BUS_IRMGR_EVENT_MAX)
```

Send Event to IARM-Bus

Once the event is registered, the application can send events to the bus. The events are always broadcast on the bus. Only the applications listening to the event will be awakened. The sending application is responsible for allocating and freeing memory used to store event data. In most cases the memory can simply be allocated from stack.

To send an IARM_BUS_IRMGR_EVENT_IRKEY an IR whenever

```
IARM_Bus_BroadcastEvent(
     IARM_BUS_IRMGR_NAME,                 /* Owner of the Event */
     IARM_BUS_IRMGR_EVENT_IRKEY,        /* Event ID from this owner */
     (void *) &eventData,                 /* IARM_Bus_IRMgr_EventData_t */
     sizeof(eventData)                        /* Length of the eventData */
 )
```

Receive Event from IARM-Bus

Any application can listen for events from sent by other applications on the IARM-Bus. In our example, an application interested in receiving Remote Keys can register for IR Manager's IARM_BUS_IRMGR_EVENT_IRKEY event. When receiving events, a handler must be supplied to process the event. This handler will be called from IARM's internal thread context.

```
IARM_Bus_RegisterEventHandler(
     IARM_BUS_IRMGR_NAME,                 /* Owner of the Event */
     IARM_BUS_IRMGR_EVENT_IRKEY,        /* Event ID from this owner*/
     _My_Event_Handler,                 /* IARM_EventHandler_t */
 )
```

An application can listen for different event with different event handlers. If a same event handler is used to listen for different events from different applications, a simple switch block can be used to further dispatch the events to their target processing.

- use **strcmp**() to dispatch event by the sending application's name.
- use **switch** to dispatch event by Event Id.

The event handler in our example would contain the following block to dispatch IR key events:

```
_My_Event_Handler(const char *owner, IARM_EventId_t eventId, void *data, size_t len)
 {
     …
     /* Dispatch By Owner first */
     if(strcmp(owner, IARM_BUS_IRMGR_NAME) == 0) {
         /* Then Dispatch by Event ID */
         switch(eventID) {
             case IARM_BUS_IRMGR_EVENT_IRKEY:
                 /*Cast data to its target type */
                  IARM_Bus_IRMgr_EventData_t * = (IARM_Bus_IRMgr_EventData_t *) data;
                 /* Processing of Data */
                 …
         }
     }
     else if(strcmp(owner, /some other application/) == 0) {
     }
     else {
     }
 }
```

Summary Of Events

If an application wants to Send events, it needs to

- Follow the naming conventions recommended.
- Register to IARM-Bus with its well-known name during initialization.

```
IARM_Result_t IARM_Bus_Init(const char name);
IARM_Result_t IARM_Bus_Connect(void);
```

- Declare its event enumerations and event data types in public header file.
- Register all its events during initialization using _**EVENT_MAX**:

```
IARM_Result_t IARM_Bus_RegisterEvent(IARM_EventId_t maxEventId);
```

- Send/Broadcast Events:

```
IARM_Result_t IARM_Bus_BroadcastEvent(const char *ownerName, IARM_EventId_t eventId, void *data, size_t len);
```

If an application wants to receive and handle events , it needs to :

- Register to IARM-Bus with its well-known name during initialization.

```
IARM_Result_t IARM_Bus_Init(const char *name);
        IARM_Result_t IARM_Bus_Connect(void);
```

- Register event handler :

```
 IARM_Result_t IARM_Bus_RegisterEventHandler(const char *ownerName, IARM_EventId_t eventId, IARM_EventHandler_t
handler);
```

- Implement the handler.

How to Create and Invoke RPC Methods

Declare Well-Known Name in Header File

Since all the RPC methods published by the application are tied to its well-known name, the application must make its well-known name available to other applications that wish to use its service. The well-known name can be thought of as the Namespace within which the application's events and RPC methods live. This name must be declared in the application's public header file. In our example, this file is irMgr.h

```
 #define IARM_BUS_IRMGR_NAME "IRMgr"

Naming Convention: IARM_BUS_<Manager>_NAME "Manager"
```

When the application initialize with IARM-Bus, it provides its well-known name:

```
 IARM_Bus_Init(IARM_BUS_IRMGR_NAME);
IARM_Bus_Connect();
```

Applications should use macro of the well-known name (IARM_BUS_IRMGR_NAME in this example) when invoking the RPC method.

Declare RPC Method Names and Argument Types in Header File

Considering the application's well-name as a namespace name, Method name is used to identify the RPC method within the namespace. Each method from a same owner has unique string name. This name must be declared in the application's public header file, so other applications can refer to this method by its method name.

In our example, IR Manager publishes two RPC Methods, that allow other applications to set and get Repeat Key Intervals. The method name is defined as:

```
#define IARM_BUS_IRMGR_API_SetRepeatInterval "SetRepeatInterval"
Naming Convention: IARM_BUS_<Manager>API<Method> "Method"
```

Applications should use macro of the method name (IARM_BUS_IRMGR_API_SetRepeatInterval in this example) when invoking the RPC method.

A RPC method is like a regular method in its declaration. It may or may not have input or output arguments. If a RPC method has any input or output argument, the data types must be declared so that other applications can invoke the method properly. The data structure should be a **C structure** that includes both input and output parameters.

The argument structure ***cannot*** have pointers. The **sizeof()** operator applied on the Param_t must equal to the actual memory allocated. Internally an equivalent of **memcpy()** is used to dispatch parameters its target. If a pointer is used in the parameters, the pointer, not the content it points to, is sent to the destination.

In our example, IR Manager declares this data structure as argument type for the **SetRepeatInterval** method.

```
typedef struct _IARM_Bus_IRMgr_SetRepeatInterval_Param_t {
     unsigned int timeoutNewValue;         /* Used when invoking RPC */
     unsigned int timeoutOldValue;         /* Used to hold return value */
 } IARM_Bus_IRMgr_SetRepeatInterval_Param_t;
```

```
Naming Convention: IARM_BUS_<Manager>_<Method>_Param_t
```

This method has an input parameter and an output parameter. It is equivalent to a non-RPC function with signature:
SetRepeatInterval (uint timeoutNewValue, uint *{}timeoutOldValue);

The RPC implementation will place the old value into the storage allocated for timeoutOldValue prior to function return.

Register RPC Methods to IARM-Bus

A RPC Method must be Registered with IARM-Bus first before it can be invoked. The application that registers the RPC method is considered the **Owner** of the method. During initialization, the application can register all the RPC methods it implements

```
IARM_Result_t IARM_Bus_RegisterCall(
     _BUS_IRMGR_API_SetRepeatInterval,         /* RPC Method Name */
     _SetRepeatInterval                        /* RPC Method Implementation*/*
)
```

Here **_SetRepeatInterval** () is the actual implementation of the RPC method. All RPC methods takes the signature of IARM_BusCall_t:

```
typedef IARM_Result_t (*IARM_BusCall_t) (void *arg);
```

The owner of the RPC Method provides its implementation. In irMgr.c, the implementation takes the following form:

```
static IARM_Result_t _SetRepeatInterval(void *arg)
{
     /* First cast the argument to its target type */
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t
     *param = (IARM_Bus_IRMgr_SetRepeatInterval_Param_t *)arg;
     /* do something */
     /* populate the output fileds of param */
 }
```

Invoke RPC Method

Once an RPC is registered, other application can invoke the method synchronously. For example, this invocation sets the key repeat interval to 200ms.

```
IARM_Bus_IRMgr_SetRepeatInterval_Param_t param;
     param. timeoutNewValue = 200;
     IARM_Bus_Call (
         IARM_BUS_IRMGR_NAME, /* Owner of the Method */
         IARM_BUS_IRMGR_API_SetRepeatInterval,/* Name of Method */
         (void *)&param, / Parameter of Method */
         sizeof(param)); /* Length of the Parameter */
)
```

Summary of RPC Methods

In summary, if an application wants to publish RPC Methods, it needs to

- Follow the naming conventions recommended.
- Register to IARM-Bus with its well-known name during initialization.

  ```
  IARM_Result_t IARM_Bus_Init(const char name);*
  IARM_Result_t IARM_Bus_Connect(void);
  ```

- Declare its RPC method names and argument types in public header file:
- Register all its RPC methods during initialization:

  ```
  IARM_Result_t IARM_Bus_RegisterCall(const char *name, IARM_BusCall_t handler);
  ```

- Invoke RPC Methods:

  ```
  IARM_Result_t IARM_Bus_Call(const char *ownerName, const char *methodName, void *arg, size_t argLen);
  ```

Decouples Applications from IARM

The RPC methods exposed via IARM can be called by multiple applications, so they are also named **Multiple-App APIs**. In contract, the regular C functions that can only be invoked from within the same Linux process that implements the C functions are named **Single-App APIs**. Sometimes, it is desirable that the application is decoupled from such API difference. In this case, the developer should provide a generic API that can be linked to either a local C method or a RPC method at build time.
In our example, IR Manager exports RPC API "SetRepeatInterval". To invoke this method, the application executes:

```
IARM_Bus_Call (
     IARM_BUS_IRMGR_NAME,                        /* Owner of the Method */
     IARM_BUS_IRMGR_API_SetRepeatInterval,       /* Name of Method */
     (void *)&param,                               / Parameter of Method */
     sizeof(param));                               /* Length of the Parameter */
)
```

This invocation injects the IARM dependency into the application. An alternative is to provide the application a generic API IRMgr_SetRepeatInterval(), whose implementation internally invokes the RPC method.

```
IRMgr_SetRepeatInterval(int newInterval)
 {
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t param;
     param.timeoutNewValue = 200;
     IARM_Bus_Call (
         IARM_BUS_IRMGR_NAME, /* Owner of the Method */
         IARM_BUS_IRMGR_API_SetRepeatInterval,/* Name of Method */
         (void *)&param, / Parameter of Method */
         sizeof(param)); /* Length of the Parameter */
     )
 }
```

Short-Term Approach

To invoke the same API, application now calls IRMgr_SetRepeatInterval().
A quick approach to decouple application from IARM is to define the generic API IRMgr_SetRepeatInterval() as inline function. In our example, we would include this in the irMgr.h.

```
#define IRMgr_SetRepeatInterval(newInterval) \
 do {
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t param;
     param. timeoutNewValue = 200;\
     IARM_Bus_Call (
         IARM_BUS_IRMGR_NAME,\ /* Owner of the Method */
         IARM_BUS_IRMGR_API_SetRepeatInterval,\/* Name of Method */
         (void *)&param,\/ Parameter of Method */
         sizeof(param));\/* Length of the Parameter */
     )
 }while(0);
```

In this approach, the application is decoupled from IARM. However, application cannot use the same API for Single-App version. This may be enough for most applications who at this point does not need a single-app version of the same API.

Long-Term Approach

If an application may use the same API's single-app version, the following changes are needed.
To invoke the same API, application now calls IRMgr_SetRepeatInterval(), which must be implemented as an actual function that has external linkage.

```
IRMgr_SetRepeatInterval(int newInterval)
 {
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t param;
     param. timeoutNewValue = 200;
     IARM_Bus_Call (
         IARM_BUS_IRMGR_NAME, /* Owner of the Method */
         IARM_BUS_IRMGR_API_SetRepeatInterval,/* Name of Method */
         (void *)&param, / Parameter of Method */
         sizeof(param)); /* Length of the Parameter */
     )
 }
```

In the IR Manager's implementation of RPC Method, we have already learned from previous sections that this RPC method is registered with.

```
IARM_Result_t IARM_Bus_RegisterCall(
     _BUS_IRMGR_API_SetRepeatInterval,  /* RPC Method Name */
     _SetRepeatInterval                      /* RPC Method Implementation*/
 )
Where _SetRepeatInterval changes the actual settings.
 static IARM_Result_t _SetRepeatInterval(void *arg)
 {
     /* First cast the argument to its target type */
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t
     *param = (IARM_Bus_IRMgr_SetRepeatInterval_Param_t *)arg;
     /* changes actual settings here */
 }
```

With generic API IRMgr_SetRepeatInterval(), we can change the _SetRepeatInterval implementation to:

```
 static IARM_Result_t _SetRepeatInterval(void *arg)
 {
     /* First cast the argument to its target type */
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t
     *param = (IARM_Bus_IRMgr_SetRepeatInterval_Param_t *)arg;
     IRMgr_SetRepeatInterval(parma->newInterval);
 }
```

Where IRMgr_SetRepeatInterval()changes the actual settings.
Now, we have at our hands two implementations of a same generic API IRMgr_SetRepeatInterval:

`The RPC Client / Multi App Version:`

```
IRMgr_SetRepeatInterval(int newInterval)
 {
     IARM_Bus_IRMgr_SetRepeatInterval_Param_t param;
     param. timeoutNewValue = 200;
     IARM_Bus_Call (
         IARM_BUS_IRMGR_NAME,                        /* Owner of the Method */
         IARM_BUS_IRMGR_API_SetRepeatInterval,       /* Name of Method */
         (void *)&param,                         /* Parameter of Method */
         sizeof(param));                         /* Length of the Parameter */
     )
 }
```

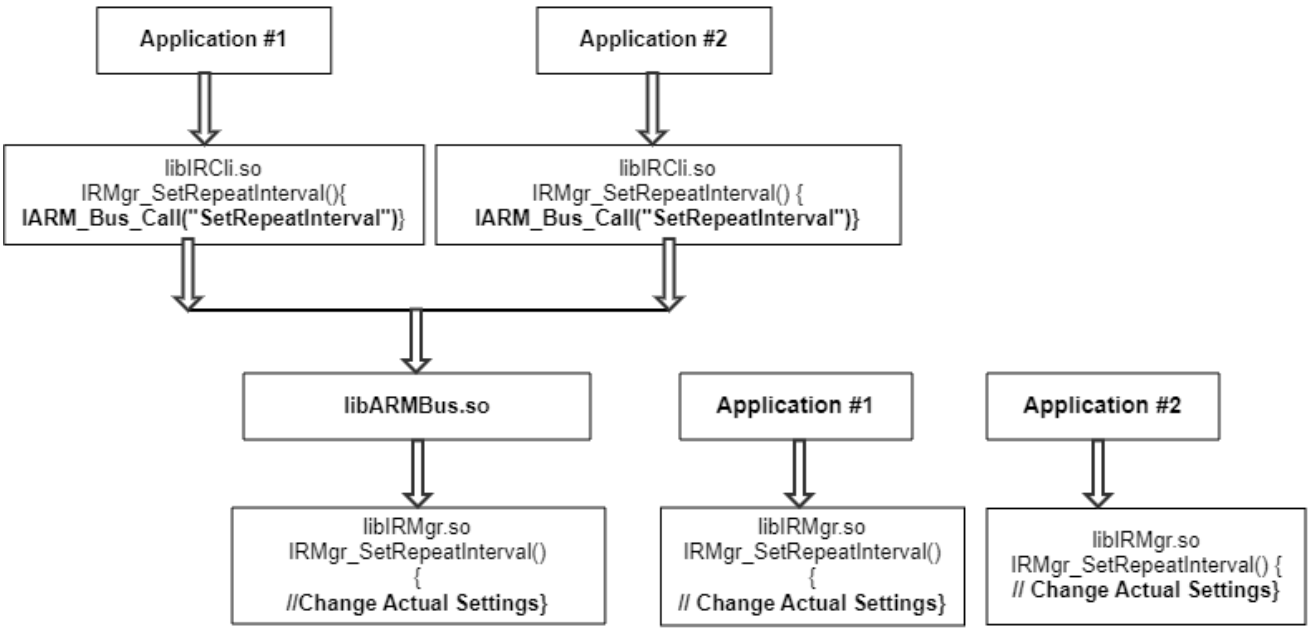
This function is linked into library libirMgrCli.so

The Manager Version/Single-App version:

```
IRMgr_SetRepeatInterval(int newInterval)
{
    /* Change the Actual Settings */
}
```

This function is linked into library libirMgr.so

The application can then choose which implementation to link to at build time based on its needs. To use the multi-app version of API, the application links to libirMgrCli.so. To use the single-app version of the API, the application links to libirMgr.so. In either choice, the application is no longer coupled to IARM. The following diagrams demonstrate the linkage relationship between applications and the generic API.



Porting layer of IARM-Bus

Most IARM Applications are not Manager Components. Rather they are connected to IARM bus only to utilize the service provided by various IARM-Bus Managers. These applications can maximum their platform independency by interfacing to only the common APIs of the IARM Bus.

Meanwhile, depending on the events being published or the dependency that an RPC method implementation may have, some part of an IARM Manager Component may be platform specific.
It is up to the developer of the manager component to craft out and abstract the platform specific portion of the manager implementation. In the IR Manager example we used, the manager receives IR signals from PIC drivers in different ways on different platforms. This difference is abstract to a PLAT_API interface defined in an internal header file plat.h, and allows most of IR Manager implementation to remain platform agnostic.

Core IARM-Bus Manager Components

By default, the IARM-Bus has the following Manager Components. The published events and RPC methods from these manager components can be found in their public header files.
On a standard system, you should see the following 3 processes running before executing your own application.

- IARMDaemonMain
- irMgrMain
- pwrMgrMain

To see the IARM managers available in the device. Use the below command,

```
ps -aef|grep Mgr*
root 462 1 0 05:07 ? 00:00:02 /usr/bin/mfrMgrMain
root 1195 1 0 05:07 ? 00:00:02 /usr/bin/sysMgrMain
root 1240 1 0 05:07 ? 00:00:02 /usr/bin/dsMgrMain
root 1343 1 1 05:07 ? 00:06:01 /usr/bin/irMgrMain
root 1545 1 0 05:07 ? 00:00:02 /usr/bin/pwrMgrMain
root 1593 1 0 05:07 ? 00:00:02 /usr/bin/deepSleepMgrMain
```

Bus Daemon

Executable name is IARMDaemonMain. The IARM Bus Daemon is a Manager Component with special privileges.

> ⚠ This Daemon component must be run before any other IARM-Bus application.

IR Manager

Executable name is irMgrMain. This manager receives IR signals from driver and dispatch it to all registered listeners on IARM Bus. The standard IR key codes sent by IR Manager are listed in core/ir/comcastIRKeyCodes.h.

Power Manager

Executable name is pwrMgrMain. This manager monitors Power IR key events and react to power state changes based on RDK Power Management Specification. It dispatches Power Mode Change events to IARM-Bus. All listeners should releases resources when entering POWER OFF/STANDBY state and reacquire them when entering POWER ON state.

Troubleshooting

Before you try out a new IARM-Application, please make sure the following components are properly running on your box:

- libIARMBus.so is at a known library path.
- Daemon process IARMDaemonMain is running.
  - ps –aef | grep IARMDaemonMain
  - You see log message "Bus Daemon HeartBeat" on console or /opt/logs/uimgr_log.txt
  - cat /opt/logs/uimgr_log.txt

```
Oct 17 08:47:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: I-ARM Bus Daemon : HeartBeat
at Thu Oct 17 08:47:00 2019
Oct 17 08:47:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: [1B blob data]
Oct 17 08:47:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: I-ARM MFR Lib: HeartBeat at Thu
Oct 17 08:47:01 2019
Oct 17 08:47:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: [1B blob data]
Oct 17 08:47:04 raspberrypi-rdk-hybrid-thunder dsMgrMain[1240]: I-ARM BUS DS Mgr: HeartBeat at Thu
Oct 17 08:47:04 2019
Oct 17 08:47:04 raspberrypi-rdk-hybrid-thunder dsMgrMain[1240]: [1B blob data]
Oct 17 08:47:06 raspberrypi-rdk-hybrid-thunder pwrMgrMain[1545]: I-ARM POWER Mgr: HeartBeat at Thu
Oct 17 08:47:06 2019
Oct 17 08:47:06 raspberrypi-rdk-hybrid-thunder pwrMgrMain[1545]: [1B blob data]
Oct 17 08:52:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: I-ARM Bus Daemon : HeartBeat
at Thu Oct 17 08:52:00 2019
Oct 17 08:52:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: [1B blob data]
Oct 17 08:52:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: I-ARM MFR Lib: HeartBeat at Thu
Oct 17 08:52:01 2019
Oct 17 08:52:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: [1B blob data]
Oct 17 08:52:04 raspberrypi-rdk-hybrid-thunder dsMgrMain[1240]: I-ARM BUS DS Mgr: HeartBeat at Thu
Oct 17 08:52:04 2019
Oct 17 08:52:04 raspberrypi-rdk-hybrid-thunder dsMgrMain[1240]: [1B blob data]
Oct 17 08:52:06 raspberrypi-rdk-hybrid-thunder pwrMgrMain[1545]: I-ARM POWER Mgr: HeartBeat at Thu
Oct 17 08:52:06 2019
Oct 17 08:52:06 raspberrypi-rdk-hybrid-thunder pwrMgrMain[1545]: [1B blob data]
Oct 17 08:57:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: I-ARM Bus Daemon : HeartBeat
at Thu Oct 17 08:57:00 2019
Oct 17 08:57:00 raspberrypi-rdk-hybrid-thunder IARMDaemonMain[234]: [1B blob data]
Oct 17 08:57:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: I-ARM MFR Lib: HeartBeat at Thu
Oct 17 08:57:01 2019
Oct 17 08:57:01 raspberrypi-rdk-hybrid-thunder mfrMgrMain[462]: [1B blob data]
```

API Documentation

To know more about SoC/Application level APIs details use in RDK, refer the link IARM BUS API Documentation