TEXAS INSTRUMENTS

Application Report

SPRAAI5 – February 2008

Using IRES and RMAN Framework Components for 'C64x+

Murat Karaorman, Gunjan Dang *Software Development Organization*

ABSTRACT

IRES is a TMS320 DSP Algorithm Standard (xDAIS) interface for management and utilization of special resource types such as hardware accelerators, certain types of memory and DMA. RMAN is a generic Resource Manager that manages software components' logical resources based on their IRES interface configuration. Both IRES and RMAN are Framework Component modules. This document presents an overview of IRES and RMAN, with examples.

Contents

1 Introduction 2
2 IRES: Standard Interface for Resource Negotiation and Utilization....3
2.1 IRES and RMAN Modules....3
2.2 IRES Interface Definition....5
2.2.1 IRES Resource Descriptors: IRES_ResourceDescriptor....6
2.2.2 IRES Interface Functions: IRES_Fxns....7
2.2.3 Generic IRES Resource Object and Handle Structures....9
2.2.4 Generic IRES Protocol Arguments....9
2.2.5 Concrete IRES Resource Interfaces....10
2.2.6 Algorithm IRES Implementation Example....15
2.3 Cooperative Multitasking and Preemption....16
2.3.1 Non-Cooperative Multi-Tasking....17
2.3.2 Cooperative Multi-Tasking – Yielding to the Same Priority....18
2.3.3 Cooperative Multi-tasking – Yielding to Higher Priority....19
3 RMAN: 'C64x+ DMA Resource Manager....21
3.1 Introduction....21
3.2 Using RMAN for Algorithm Integration....21
3.3 IRESMAN: Standard Interface for Implementing Device-specific Resource Managers....23
3.3.1 IRES Resource Description ires_<resource>.h....23
3.3.2 IRESMAN Resource Manager Description iresman_<resource>.h....23
3.4 RMAN Configuration....26
3.4.1 XDC Configuration Parameters....26
3.4.2 Low-level "C" Configuration Parameters....28
3.5 RMAN Configuration Examples....30
3.5.1 Static and Dynamic Registration of Resource Managers with RMAN....30
3.5.2 Configuring RMAN to use DSKT2....30
3.5.3 Configuring RMAN to Not Use DSKT2....31
3.6 Other Configuration Parameters....31
3.7 Configuring RMAN Without Using RTSC....32
3.8 Configuring RMAN Using RTSC Tooling....32
3.9 RMAN Version support....33

3.10 RMAN Functions 33
3.10.1 RMAN_exit....34
3.10.2 RMAN_init....35
3.10.3 RMAN_register 35
3.10.4 RMAN_unregister 36
3.10.5 RMAN_assignResources....36
4 References 40

Figures

Figure 1. IRES Function Calling Sequence 5
Figure 2. IALG and IDMA3 Interfaces and Application Implemented with Framework Components 6
Figure 3. Non-Cooperative Multitasking....18
Figure 4. Cooperative Multi-tasking: Yielding to the Same Priority 19
Figure 5. Cooperative Multi-tasking: Yielding to a Higher Priority 20
Figure 6. IRESMAN Functions Calling Sequence....24
Figure 7. RMAN Functions Calling Sequence....34

Tables

Table 1. IRES Functions....4
Table 2. IRES_ResourceDescriptor....4
Table 3. RMAN Functions 4

1 Introduction

Media applications require multiple algorithms to run concurrently and share available system resources (CPU, Memory, DMA, Hardware Accelerators, etc.) with high utilization rates. Application frameworks must ensure minimal overhead for context switching, algorithm/resource initialization, activation and deactivation to meet the real-time, scheduling and quality-of-service requirements.

The TMS320 DSP Algorithm Standard (xDAIS) defines standard interfaces for algorithms to request and acquire resources from the application framework. The standard also defines properties and types of resources, along with rules and guidelines for their use, initialization, and sharing.

IALG is the primary interface that xDAIS algorithms must implement. The IALG interface defines a standard “memory” resource type with specific size, alignment, space, and physical address attributes, and defines how the algorithm accesses memory through IALG-defined query, initialization, relocation, activation and deactivation stages. Standard IALG-specified functions enable the application framework to statically or dynamically query an algorithm’s instance memory requirements, allocate and assign the requested “*memory*” and use the algorithm instance to perform its specific functions.

The IDMA2 and IDMA3 interfaces were introduced as optional secondary xDAIS interfaces to manage and utilize “DMA” resources, but did not support some types of physical or logical resources, such as hardware accelerators, non-IALG memory types or certain DMA resources. IRES is a new generic, extendible interface that supports new resource types. Algorithms must still implement the IALG interface for instance creation, activation and deactivation, and may implement IRES as an optional, secondary resource management and utilization interface for new resource types.

This document presents an overview of the IRES interface, along with some concrete resource types and resource managers that illustrate the definition, management and use of new types of resources.

2 IRES: Standard Interface for Resource Negotiation and Utilization

2.1 IRES and RMAN Modules

IRES is a generic, resource-agnostic, extendible resource query, initialization and activation interface. The application framework must define, implement and support concrete resource interfaces in the form of IRES extensions. Each algorithm implements the “generic” IRES interface to request one or more “concrete” IRES resources.

IRES defines standard interface functions that the framework uses to query, initialize, activate/deactivate and reallocate concrete IRES resources. To create an algorithm instance within an application framework, the algorithm and the application framework must agree on the concrete IRES resource types that are being requested. The framework must call the IRES interface functions, in addition to the IALG functions, to perform IRES resource initialization, activation and deactivation.

The IRES interface also introduces support for a new standard protocol for cooperative pre-emption, in addition to the IALG-style non-cooperative sharing of scratch resources. Cooperative preemption allows activated algorithms to yield to higher priority tasks sharing common scratch resources.

Framework Components includes the following modules and interfaces to support algorithms requesting IRES-based resources:

- **IRES.** This is the standard interface allowing the client application to query and provide the algorithm with its requested IRES resources.
- **RMAN.** This is the generic IRES-based resource manager. It manages and grants concrete IRES resources to requesting algorithms and applications. RMAN uses a new standard interface, the IRESMAN, to support runtime registration of concrete IRES resource managers.

Client applications call the algorithm’s IRES interface functions to query its concrete IRES resource requirements. If the requested IRES resource type matches a concrete IRES resource interface supported by the application framework, and a resource is available, then the client grants the algorithm logical IRES resource handles representing the granted resources. Each granted handle provides the algorithm with access to the resource as defined by the concrete IRES resource interface.

The following tables summarize the API functions and structures used by the IRES and RMAN interfaces.

Table 1. IRES Functions

Functions	Description
getResourceDescriptors ()	Query function to obtain the list of IRES resources requested by the algorithm instance.
numResourceDescriptors()	Query function to obtain the number of IRES resources requested.
initResources()	Assignment function to grant the algorithm instance the list of IRES resources it requested. The algorithm can initialize internal instance memory with resource information, but may not use or access the resource state until the resource is activated via the activateResource call.
reinitResources()	Re-assignment function to grant the algorithm instance a list of "modified" IRES resources.
deinitResources()	De-initialization function to revoke the resources that have been granted to the algorithm instance.
activateResource()	Resource activation call to grant the algorithm instance exclusive access to a potentially shared resource. The algorithm can now access, initialize and/or restore from a context saved during previous deactivation to use the resource.
activateAllResources()	Resource activation call to grant the algorithm instance exclusive access to all resources it acquired via IRES. Algorithm can now access, initialize and/or restore from a context saved during previous deactivation to use the resources.
deactivateResource()	Resource deactivation call to revoke the algorithm instance's exclusive access to the potentially shared resource.
deactivateAllResources()	Resource deactivation call to revoke the algorithm instance's exclusive access to ALL shared resources. Algorithm must save any context that is needed to restore the state during the next resource activation call.

Table 2. IRES_ResourceDescriptor

Structure Fields	Description
resourceName	String containing the package name to identify the resource.
protocolArgs	Pointer to the Resource Protocol Arguments. The Resource Manager selects the appropriate Resource Protocol based on the supplied "resourceName", and uses the protocol to construct the IRES Resource Handle.
revision	The revision of the IRES_ResourceProtocol Interface expected by the client algorithm.
handle	The handle to the object representing the requested resource. The handle is initially set to 'null' by the requesting algorithm. The Resource Manager allocates the resource and constructs the handle.

Table 3. RMAN Functions

Functions	Description
RMAN_init	Initialize the RMAN object with static information from the headers/configuration.
RMAN_exit	Finalization method of the RMAN module.
RMAN_register	Register the protocol/protocol revision and the device-specific resource manager implementation with the Resource Registry of RMAN.
RMAN_unregister	Unregister the protocol and the corresponding resource manager implementation from the Resource Registry.
RMAN_assignResources	Query, allocate and assign the resources the algorithm requested via IRES interface.

Functions	Description
RMAN_freeResources	Free resources held by the algorithm resource handle.
RMAN_activateResource	Activate a particular resource held by the algorithm.
RMAN_activateAllResources	Activate all resources held by the algorithm.
RMAN_deactivateResource	Deactivate a particular resource held by the algorithm.
RMAN_deactivateAllResources	Deactivate all resources held by the algorithm.

2.2 IRES Interface Definition

The IRES interface is implemented by algorithms that need concrete resources defined by extending the IRES interface. The application framework Resource Manager calls the algorithm's IRES interface functions to query and subsequently allocate and grant the requested resources. RMAN is one such reference resource manager delivered as part of TI's Framework Components package, however, applications may choose to supply their own IRES compliant resource managers.

An algorithm implements the IRES interface by defining and initializing a global structure of type IRES_Fxns. Every function defined in this structure must be implemented and assigned to the appropriate field in the structure. Figure 1 illustrates the calling sequence for IRES functions, and how they relate to the IALG functions executed during algorithm instance creation and real-time operation.

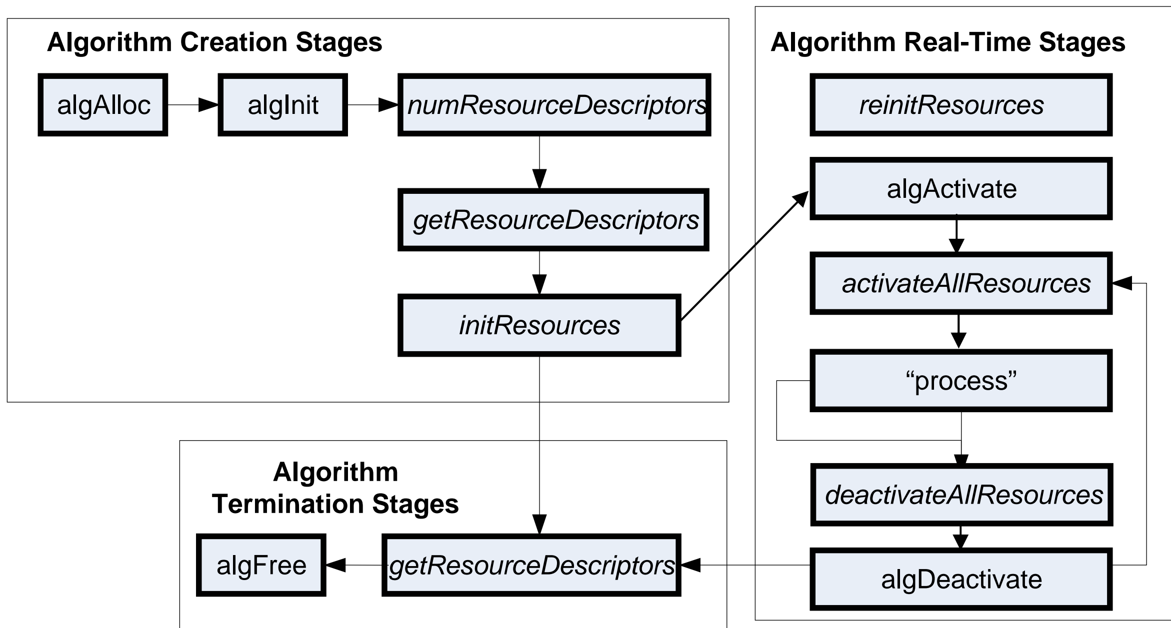

Figure 1. IRES Function Calling Sequence

The getResourceDescriptors() and reinitResources() functions can be called at any time in the algorithm's real-time stages. The algMoved() and algNumAlloc() functions were omitted from this figure for simplicity.

The getResourceDescriptors() and initResources() functions must be called after IALG::algInit(). The numResourceDescriptors() function, called here after algInit(), can be called before the algorithm instance object is created if the framework wants to query the algorithm about its IRES resource requirements before creating the instance object.

Note: Framework Components provides an IRES resource manager, RMAN, that provides APIs to grant resources to algorithms by calling their IRES functions. This is discussed in Section 2.1.

Figure 2 illustrates a typical system with an algorithm implementing the IALG and IRES interfaces and the application with the Framework Components RMAN resource manager.

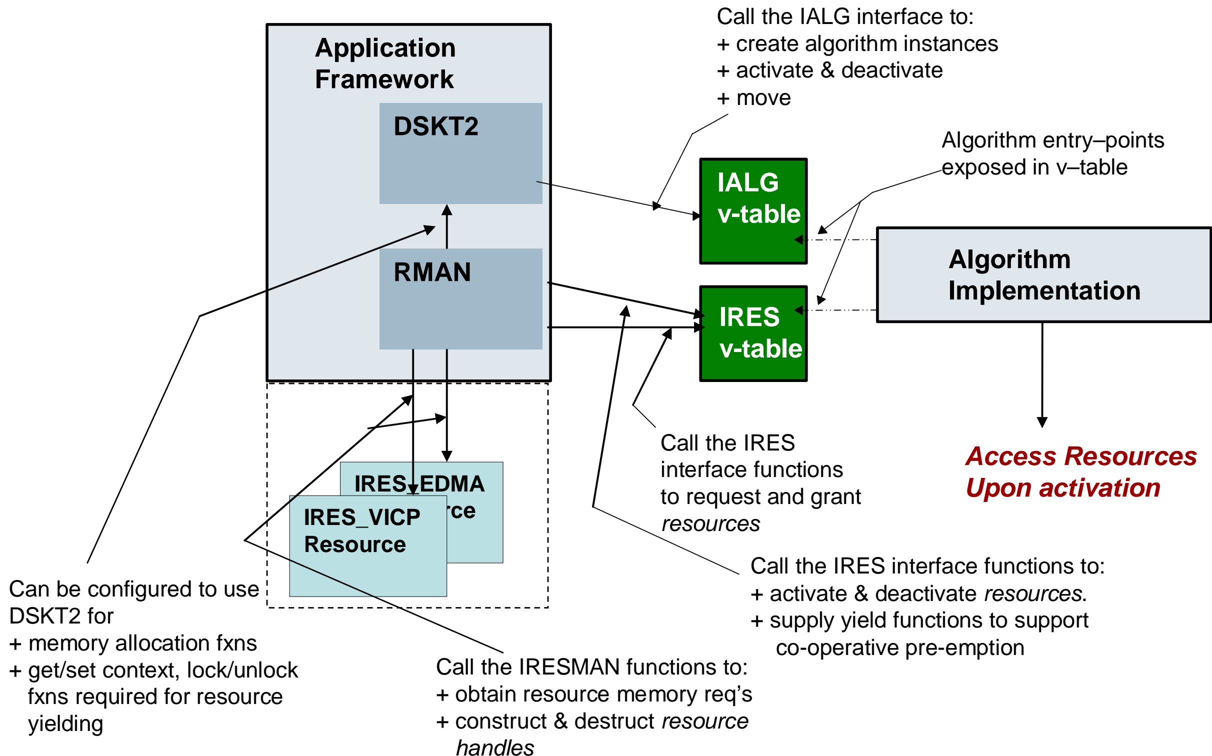

Figure 2. IALG and IDMA3 Interfaces and Application Implemented with Framework Components

2.2.1 IRES Resource Descriptors: IRES_ResourceDescriptor

The IRES functions use the IRES_ResourceDescriptors to characterize each logical IRES resource granted to the requesting algorithm or module. The Resource Manager utilizes the information in the resource descriptor to allocate the resource, and then passes its handle to the algorithm instance.

The IRES_ResourceDescriptor structure has the following fields:

String	resourceName
IRES_ProtocolArgs	*protocolArgs
IRES_ProtocolRevision	*revision
IRES_Obj	*handle

This list describes these fields:

- **String resourceName**

  String containing the package name to identify the resource.

- **IRES_ProtocolArgs *protocolArgs**

  Pointer to the Resource Protocol Arguments. The Resource Manager selects the appropriate Resource Protocol based on the supplied "resourceName", and uses the protocol to construct the IRES Resource Handle.

- **IRES_ProtocolRevision *revision**

  The revision of the IRES_ResourceProtocol Interface expected by the client algorithm.

- **IRES_Obj *handle**

  The handle to the object representing the requested resource. The handle is initially set to 'null' by the requesting algorithm. The Resource Manager allocates the resource and constructs the handle.

2.2.2 IRES Interface Functions: IRES_Fxns

The application framework calls IRES interface functions (IRES_Fxns) to query and grant resources requested by the algorithm at initialization time, and to make changes to these resources at run-time.

Void* IRES_Fxns::implementationId	
IRES_Status(* IRES_Fxns::getResourceDescriptors)(	IALG_Handle handle, IRES_ResourceDescriptor *resourceDescriptors)
Int32(* IRES_Fxns::numResourceDescriptors)(	IALG_Handle handle)
IRES_Status(* IRES_Fxns::initResources)(	IALG_Handle handle, IRES_ResourceDescriptor *resourceDescriptor, IRES_YieldFxn yieldFxn, IRES_YieldArgs yieldArgs)
IRES_Status(* IRES_Fxns::reinitResources)(	IALG_Handle handle, IRES_ResourceDescriptor *resourceDescriptor, IRES_YieldFxn yieldFxn, IRES_YieldArgs yieldArgs)
IRES_Status(* IRES_Fxns::deinitResources)(	IALG_Handle handle, IRES_ResourceDescriptor *resourceDescriptor)
IRES_Status(* IRES_Fxns::activateResource)(	IALG_Handle handle, IRES_Handle resourceHandle)
IRES_Status(* IRES_Fxns::activateAllResources)(	IALG_Handle handle)
IRES_Status(* IRES_Fxns::deactivateResource)(	IALG_Handle handle, IRES_Handle resourceHandle)
IRES_Status(* IRES_Fxns::deactivateAllResources)(	IALG_Handle handle)

The IRES interfaces functions are described in the following list:

- **implementationId.** Unique pointer that identifies the module implementing this interface.
- **getResourceDescriptors.** Query function to obtain the list of IRES resources requested by the algorithm instance.
- **numResourceDescriptors.** Query function to obtain the number of IRES resources requested by the algorithm instance, which is also the number of resource descriptors that must be passed to the getResourceDescriptors() function.
- **initResources.** Assignment function to grant the algorithm instance the list of IRES resources it requested. The algorithm can initialize internal instance memory with resource information, but may not use or access the resource state until the resource is activated via the activateResource call.
- **reinitResources.** Re-assignment function to grant the algorithm instance a list of "modified" IRES resources. The algorithm may choose to not support the re-assignment and indicate this by returning failure status. In case of success the algorithm updates its internal state to reflect the new resource information, but may not use or access the resource state until the resource is activated via the activateResource call.
- **deinitResources.** Deinitialization function to revoke the resources that were granted to the algorithm instance.
- **activateResource.** Resource activation call to grant the algorithm instance exclusive access to the potentially shared resource. Once activated, the algorithm can access, initialize and/or restore from a previous context saved during deactivation. Activating each resource individually optimizes resource management by the framework.
- **activateAllResources.** Resource activation call to grant the algorithm instance exclusive access to all resources it acquired via IRES. Once activated, the algorithm can access, initialize and/or restore from a previous context saved during deactivation.
- **deactivateResource.** Resource deactivation call to revoke the algorithm instance's exclusive access to the potentially shared resource. Algorithm must save any context that is needed to restore the state during the next resource activation call.
- **deactivateAllResources.** Resource deactivation call to revoke the algorithm instance's exclusive access to ALL shared resources. The algorithm must save any context that is needed to restore the state during the next resource activation call.

2.2.3 Generic IRES Resource Object and Handle Structures

Each IRES resource object holds the private state associated with the resource. The application framework resource manager creates and initializes this state with the physical or logical attributes of the concrete resource type that has been allocated. The Concrete IRES Resource interface defines the concrete resource object properties.

When the resource is created with its persistent field set to “false”, the resources assigned to the handle are considered to be “scratch”, as the definition applies to IALG memory attributes. Algorithms must initialize the resource state each time they are put in an “active” state, via an IRES activation call (following IALG::algActivate), and must save any necessary channel context when they are deactivated.

The IRES_Obj structure has the following fields:

Int32	persistent
Void(* *function*)()	getStaticProperties

This list describes these fields:

- **Int32 persistent**

  Indicates if the resource has been allocated as persistent or scratch.

- **Void(* getStaticProperties)(struct IRES_Obj *resourceHandle, IRES_Properties *resourceProperties)**

  Obtains static properties associated with this resource, such as the device’s register layer.

2.2.4 Generic IRES Protocol Arguments

The abstract protocol arguments structure defines the resource independent fields common to all IRES resource types. Each algorithm will supply, as part of the extended IRES resource descriptor definition, a structure containing the attributes of the type of concrete IRES resource that it is requesting. The extended IRES interface of the concrete resource type defines the resource-specific attributes extending the IRES_ProtocolArgs structure definition. The structure containing the actual arguments must be allocated by the algorithm as part of its IALG instance memory.

The IRES_ProtocolArgs structure has the following fields:

Int32	size
IRES_RequestMode	mode

This list describes these fields:

- **Int32 size**

  Size of this structure in bytes.

- **IRES_RequestMode mode**

  The mode can be either IRES_SCRATCH or IRES_PERSISTENT. When persistent, the resource will be allocated exclusively for this algorithm.

2.2.5 Concrete IRES Resource Interfaces

The abstract IRES interface is convenient for defining a resource-agnostic communication protocol through which algorithms can request and receive resources. However, algorithms will always request resources with a well-defined IRES-based interface. The abstract IRES layer allows the algorithm to simply use a “name”, “revision”, and “mode” to identify the concrete resource it is requesting. The algorithm will also supply the resource protocol arguments, as defined by the extended concrete resource interface.

The framework’s responsibility is to ‘locate’ the type of request designated by the algorithm’s IRES resource descriptor, and if the name and revision matches any of the concrete IRES resource types it supports, then it can parse the concrete resource protocol arguments supplied in the resource descriptor, allocate the resource and give the algorithm the handle using the generic IRES functions.

Algorithms and frameworks can define and introduce new resource types, or use supported concrete resource types which may be pre-defined by standard suppliers, such as Texas Instruments Framework Components. This section presents two concrete IRES resource interfaces defined and supported by TI Framework Components: IRES_EDMA3CHAN and IRES_HDVICP. Additional resource types have been introduced and documented by standard TI Framework Components releases.

Each concrete IRES resource interface definition may supply the following extensions to the abstract IRES interface to define:

- A name representing the new resource type.
- Revision information with each resource interface, to support evolution of the concrete resource types.
- The resource-specific protocol arguments that the algorithm uses to characterize the resource it is requesting.
- The properties of the specific resource instance that is allocated and passed to the algorithm via the IRES handle. This means extending the IRES Object definition, introducing the attributes of the allocated resource that are made visible via the extended IRES handle.
- The common, static properties of the concrete resource. These are the attributes of the resource that are made visible by the new resource definition that do not depend on the resource protocol arguments that were used to request the resource with, for example, the base register addresses of the hardware accelerator resource.
- A functional interface for the resource. This is not required, but optionally the extended static property of the resource interface can define a functional interface that the algorithm uses to perform operations on the resource.
- Additional resource-specific types and structures that can be used by the algorithm to request and use the resource.

2.2.5.1 EDMA3 Concrete Resource Interface

The IRES EDMA3 Resource Interface, IRES_EDMA3CHAN, allows algorithms to request and receive handles representing EDMA3 resources associated with a single EDMA3 channel. This is a very low-level resource definition specifically for the ‘C64x+ EDMA3 controller. Note that the existing xDAIS IDMA3 and IDMA2 interfaces can still be used to request logical DMA channels, but the IRES EDMA3CHAN interface provides the ability to request resources with finer precision than with IDMA2 or IDMA3.

2.2.5.1.1 EDMA3 Channel Protocol Arguments

The following attributes are used by the algorithm when requesting an IRES_EDMA3CHAN resource:

int	size
IRES_RequestMode	mode
short	numPaRams
short	paRamIndex
short	numTccs
short	tccIndex
short	qdmaChan
short	edmaChan
short	contiguousAllocation
short	shadowPaRamsAllocation

The field definitions are described in the following list:

- **int size**

  Size of this structure in bytes.

- **IRES_RequestMode mode**

  The mode can be either IRES_SCRATCH or IRES_PERSISTENT. When persistent, the resource will be allocated exclusively for this algorithm.

- **short numPaRams**

  Number of EDMA3 Parameter RAMs (PaRAMs) requested. Can request 0 to IRES_EDMA3CHAN_MAXPARAMS many PaRams with a single request descriptor.

- **short paRamIndex**

  Describes the type of paRams required. Can specify the start Index of a particular PaRam block (if numPaRams > 1 and requesting contiguous PaRams) or request any PaRAM block, via selecting IRES_EDMA3CHAN_PARAM_ANY.

- **short numTccs**

  The number of TCC(s) requested: 0 to IRES_EDMA3CHAN_MAXTCCS.

- **short tccIndex**

  Type of TCCs required. Can either specify the start Index of the TCC (if numPaRams > 1 and requesting contiguous TCCs), or request any TCC via IRES_EDMA3CHAN_TCC_ANY.

- **short qdmaChan**

  QDMA channel number that is being requested. Either a specific QDMA channel, or "any" available channel can be requested via IRES_EDMA3CHAN_QDMACHAN_ANY, otherwise select "none" via IRES_EDMA3CHAN_CHAN_NONE.

- **short edmaChan**

  EDMA channel number that is being requested. Either a specific EDMA channel, or "any" available channel can be requested via IRES_EDMA3CHAN_EDMACHAN_ANY, otherwise select "none" via IRES_EDMA3CHAN_CHAN_NONE.

- **short contiguousAllocation**

  Flag indicating if contiguous allocation of PaRams and TCCs is required.

- **short shadowPaRamsAllocation**

  Flag indicating if additional memory should be allocated as part of the IRES handle object, which may be used for shadowing the physical PaRams.

2.2.5.1.2 EDMA3 Resource Properties

The following fields are introduced as an extension of the IRES_Obj structure:

IRES_Obj	ires
IRES_EDMA3CHAN_PaRamStruct	*shadowPaRams
unsigned int	*assignedPaRamAddresses
short	*assignedPaRamIndices
short	*assignedTccIndices
short	assignedNumPaRams
short	assignedNumTccs
short	assignedQdmaChannelIndex
short	assignedEdmaChannelIndex
unsigned int	esrBitMaskL
unsigned int	esrBitMaskH
unsigned int	iprBitMaskL
unsigned int	iprBitMaskH

The IRES extension fields are described in the following list:

- **IRES_EDMA3CHAN_PaRamStruct *shadowPaRams**

  Pointer to memory allocated for shadowing PaRams, if requested.

- **unsigned int *assignedPaRamAddresses**

  Physical Addresses of assigned PaRams.

- **short *assignedPaRamIndices**

  Indices of the assigned PaRams.

- **short *assignedTccIndices**

  Indices of the assigned TCCs.

- **short assignedNumPaRams**

  Number of assigned PaRams.

- **short assignedNumTccs**

  Number of assigned TCCs.

- **short assignedQdmaChannelIndex**

  Assigned QDMA channel index, if requested.

- **short assignedEdmaChannelIndex**

  Assigned EDMA channel index, if requested.

- **unsigned int esrBitMaskL**

  Event set register bit masks to trigger EDMA3 transfer on the allocated channel, if requested.

- **unsigned int esrBitMaskH**

  Event set register bit masks to trigger EDMA3 transfer on the allocated channel if requested.

- **unsigned int iprBitMaskL**

  Interrupt pending register bit mask to pend on transfer completion.

- **unsigned int iprBitMaskH**

  Interrupt pending register bit mask to pend on transfer completion.

Many of these fields are very specific in their definition and usage, and they are mentioned here as reference only. Please see the API reference for the specific revision of the interface definition.

2.2.5.1.3 EDMA3 Static Resource Properties

The following attributes are defined by the IRES_EDMA3CHAN_Properties structure to represent the static properties of the EDMA3 peripheral, that is exposed to the algorithm instance through the assigned resource handle's **getStaticProperties()** function pointer.

- **numDmaChannels**. Total number of DMA channels supported by the EDMA3 Controller.
- **numQdmaChannels**. Total number of QDMA channels supported by the EDMA3 Controller.
- **numTccs**. Total number of TCCs supported by the EDMA3 Controller.

- **numPaRAMSets.** Number of PaRAM Sets supported by the EDMA3 Controller.
- **numEvtQueue.** Number of Event Queues in the EDMA3 Controller.
- **numTcs.** Number of Transfer Controllers (TCs) in the EDMA3 Controller.
- **numRegions.** Number of Regions on this EDMA3 controller.
- **dmaChPaRAMMapExists.** Channel mapping configuration. A value of 0 (NO channel mapping) implies that there is fixed association for a channel number to a parameter entry number. In other words, PaRAM entry n corresponds to channel n.
- **memProtectionExists.** Boolean indicating existence or lack of a memory protection feature.
- **IRES_EDMA3CHAN_EDMA3RegisterLayer *globalRegs.** Pointer to Global Register Region of EDMA3 Channel Controller Registers. The definition of this structure corresponds to the physical layout of the registers that can be accessed as named fields of the structure definition.

2.2.5.2 HDVICP Hardware Accelarator Concrete Resource Interface

The IRES HDVICP Resource Interface, IRES_HDVICP, allows algorithms to request and receive handles representing Hardware Accelerator resource, HDVICP, on supported hardware platforms, such as DM6467. For example, on DM6467, where there are two HDVICP co-processor subsystems, algorithms can request and acquire one of the co-processors using a single IRES request descriptor.

IRES_HDVICP is an example of a very simple resource type definition, which operates at the granularity of the entire processor and does not publish any details about the resource that is being acquired other than the ‘id’ of the processor. It leaves it up to the algorithm to manage internals of the resource based on the ‘id’.

2.2.5.2.1 HDVICP Protocol Arguments

The following attributes are used by the algorithm when requesting an IRES_HDVICP resource:

int	base.size
IRES_RequestMode	base.mode
IRES_HDVICP_RequestType	id

The field definitions are described in the following list:

- **int size**

  Size of this structure in bytes.

- **IRES_RequestMode mode**

  The mode can be either IRES_SCRATCH or IRES_PERSISTENT. When persistent, the resource will be allocated exclusively for this algorithm.

- **IRES_HDVICP_RequestType id**

  IRES_HDVICP_RequestType can be one of the following co-processor types:

  – IRES_HDVICP_ID_0 -> HDVICP processor 0

  – IRES_HDVICP_ID_1 -> HDVICP processor 1

  – IRES_HDVICP_ID_ANY -> Any available HDVICP processor

On DM6467, these processors are not symmetric, and HDVICP processor-0 has more hardware capabilities for 'video encoding', so algorithms can request the right processor type based on the functional requirements.

2.2.5.2.2 HDVICP Resource Properties

The following fields are introduced as an extension of the IRES_Obj structure:

- **Void *info**

  Currently undefined.

- **Int id**

  The ID of the HDVICP coprocessor that has been granted. Must be either IRES_HDVICP_ID_0 or IRES_HDVICP_ID_1.

2.2.6 Algorithm IRES Implementation Example

1. Use the supported IRES interface header to request, for example, an HDVICP resource:

```
#include <ti/sdo/fc/ires/hdvicp/ires_hdvicp.h>
```

2. Allocate instance memory (via IALG) for each IRES request descriptor's protocol argument structure. In this example, the memory allocated is defined as the IALG instance object's structure's member, protocolArgs, which is an array of type IRES_HDVICP_ProtocolArgs.

3. Implement the IRES::getRequestDescriptors() function, and fill out an IRES resource descriptor for an HDVICP processor resource:

```
resourceDescriptors[0].resourceName = IRES_HDVICP_PROTOCOLNAME;
resourceDescriptors[0].revision  = &IRES_HDVICP_PROTOCOLREVISION;

/*  Fill the HDVICP protocol args to request:
 *   - Either processor 0 or 1 (i.e. ANY)
 *   - as scratch HDVICP resource
 */
algHandle->protocolArgs[0].base.size =
                           sizeof(IRES_HDVICP_ProtocolArgs);
algHandle->protocolArgs[0].base.mode = IRES_SCRATCH;
algHandle->protocolArgs[0].id        = IRES_HDVICP_ID_ANY;
/* Pass the request args in the descriptor */
resourceDescriptors[0].protocolArgs =
                (IRES_ProtocolArgs *)&(algHandle->protocolArgs);
```

4. Implement the IRES::initResources() function, where the algorithm instance receives the allocated resource handles and saves them for its use.
   - Allocate space in instance memory to save resource handles and any related state information.
   - Save any framework-supplied 'yield' function pointer & args, should the codec choose to support "cooperative preemption" by yielding. (Optional.)

```
algHandle->hdvicp =
            (IRES_HDVICP_Handle)resourceDescriptor[0].handle;
/* Obtain the actual ID of HDVICP Resource granted */
algHandle->hdvicpID = (IRES_HDVICP_RequestType)
          ((IRES_HDVICP_Handle)resourceDescriptor[0].handle)->id;
/* Save the Yield function pointer and arguments */
algHandle->yieldFxn =   yieldFxn;
algHandle->yieldArgs =  yieldArgs;
```

5. Implement the IRES::activateAllResources() function, where the algorithm performs necessary initialization of any volatile state associated with the resource, e.g.:
   - Reset and load the HDVICP ARM968 & other subsystems.
   - Set up the IPC between GEM and HDVICP.
6. Implement IRES::deactivateAllResources() function.

   This is where algorithm does all the necessary work to save any volatile state associated with the resource that it will need later to activate it during the next IRES activation stage.

2.3 Cooperative Multitasking and Preemption

A significant enhancement introduced by the IRES interface is to define a standard protocol for algorithms' cooperation with their multi-tasked or preemptive operating frameworks to support efficient context switching and resource sharing.

The primary motivation behind this enhancement is to enable demanding media applications which require multiple algorithms to run concurrently and share available system resources (CPU, Memory, DMA, Hardware Accelerators, etc.) with high utilization rates. Application frameworks need to ensure minimal overhead for context switching, algorithm/resource initialization, activation and deactivation, otherwise they may not meet the real-time, scheduling and quality-of-service requirements.

A majority of current xDAIS/xDM-compliant codecs process data streams in large-granularity chunks (e.g. full video-frame) during which they cannot be pre-empted. Framework context switches can only occur at frame boundaries, resulting in long periods of non-preemptible operation that may lead to problems at the application level: long latency and response times, priority inversions and under-utilization of resources, which may cause problems with quality and real-time operation.

Without standard interfaces for cooperation, typical frameworks may not support unconstrained pre-emption for common architectures due to hardware limitations to do context save/restores on shared resources (e.g. EDMA3, VICP/IMCOP). Additionally, even when the hardware provides support for preemption, the context save/restores may be too expensive without algorithm-specific or application-specific knowledge.

Changing the granularity of data for processing requires API changes (e.g. to support processing frames vs. frame-slices) which impacts both algorithm and application development schedules and complexity.

IRES introduces support for cooperative preemption by specifying a standard way for frameworks to supply algorithms a “yield” function pointer and arguments. In return, the algorithms are encouraged to cooperate by calling this framework supplied “yield” function at points of its own choosing, during its “active processing” stages. When the algorithm calls the yield function, it supplies the state-dependent, efficient context save and restore functions, which the framework calls only when there is a real context switch. The algorithm, for example a video encoder, may transparently “yield” at “slice” boundaries, allowing context switches/preemption to occur at sub-frame granularity, while allowing the framework to operate on a full-frame granularity.

The following subsections describe example event sequences for non-cooperative and cooperative multitasking.

2.3.1 Non-Cooperative Multi-Tasking

Without the ability to yield, algorithms must run to completion once they have been activated and their ‘processing’ interface functions are called. The timing diagram in Figure 3 illustrates, as explained below:

1. The framework application starts processing in the context of Thread. Event 1 corresponds to an asynchronous arrival of a request to perform a ‘process’ operation on algorithm instance, A. In this example, Thread 1 is ready to run, so it starts processing by first performing an IALG::Activation of algorithm instance A, then calls *A’s process function, which runs to completion without issuing “yield.”*
2. Event 2 corresponds to the framework’s receipt of an asynchronous request, using the second thread, Thread 2, to call ‘process’ operation using algorithm instance B. At this point, Thread 2 cannot be scheduled to run, since Thread 1, which is same or higher priority, is actively running.
3. Thread 1, algorithm instance A, runs to completion and exits or blocks in application framework-defined state (e.g. RMS layer in Bridge, Codec Engine Frameworks).
4. The O/S context switches and Thread 2 can run. Thread 2 subsequently activates, runs and deactivates algorithm instance B.
5. Thread 2 exits or blocks in the framework layer, awaiting next processing request to arrive.

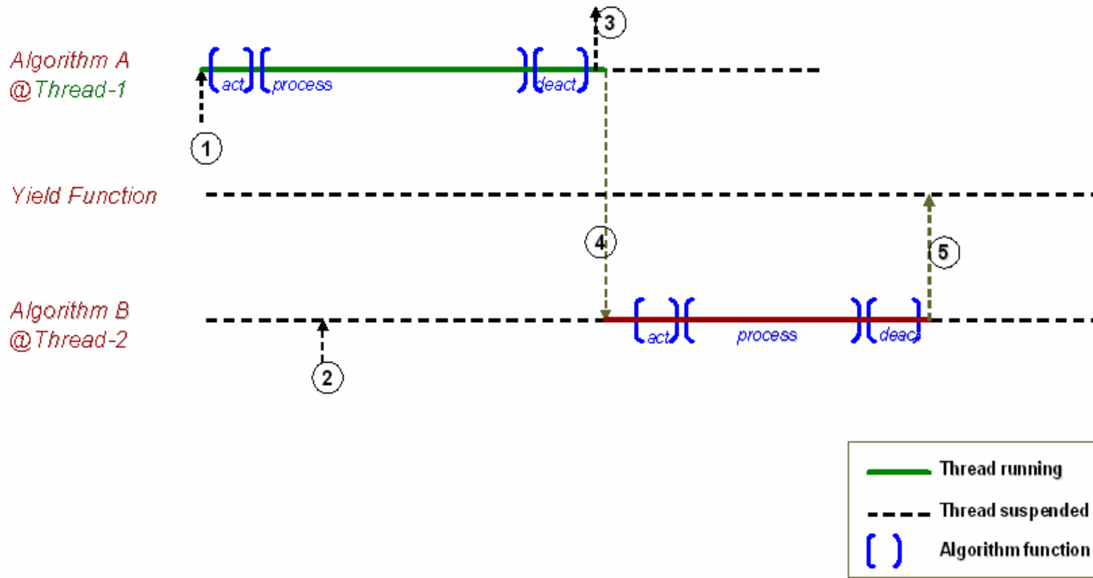

Figure 3. Non-Cooperative Multitasking

2.3.2 Cooperative Multi-Tasking – Yielding to the Same Priority

The timing diagram in Figure 4 and the steps below describe the event where two threads yield to the same priority:

1. The framework application starts processing in the context of Thread. Event 1 corresponds to an asynchronous arrival of a request to perform a ‘process’ operation on algorithm instance, A. In this example Thread 1 is ready to run, so it starts by acquiring a group resource lock, which succeeds. It then activates algorithm instance A, and calls *A’s process function.*
2. The framework receives an event indicating a request for Thread 2 to run. At this point, Thread 2 cannot be scheduled to run, since Thread 1, which is same or higher priority, is actively running.
3. Thread 1’s algorithm A, inside its process() function calls IRES Yield.
4. Thread 1, executing the framework supplied Yield() function, releases the group-lock, and issues a blocking O/S function (such as TSK_yield in DSP BIOS).
5. Thread 1 is blocked and O/S context switches to the same priority available thread, 2.
6. Thread 2 starts running, acquires group-lock, checks the ‘yield context’, and discovers that there is a “yielding” algorithm, A. Therefore it saves A’s context by calling the Yield-Context supplied_A::contextSave().
7. Thread 2 activates algorithm instance B, and then calls its process() function.
8. When Algorithm B’s process() returns, Thread 2 deactivates B, releases the group-lock, and exits or blocks awaiting next request. When Thread 1 is blocked, the O/S context switches to resume the blocked thread, Thread 1.

9. Thread 1 resumes inside the Yield() function, attempts to acquire the group-lock, which succeeds since algorithm B has released the lock in (8). At this point, the Yield function discovers that a context switch has occurred, so it restores its own algorithm A's context by calling Yield Context supplied A::contextRestore.

10. Thread 1, upon restoring A's context, readies to resume.

11. Thread 1 Yield() function returns, resuming A::process() to the point where it had issued Yield.

12. Thread 1 releases group-lock, exits or blocks.

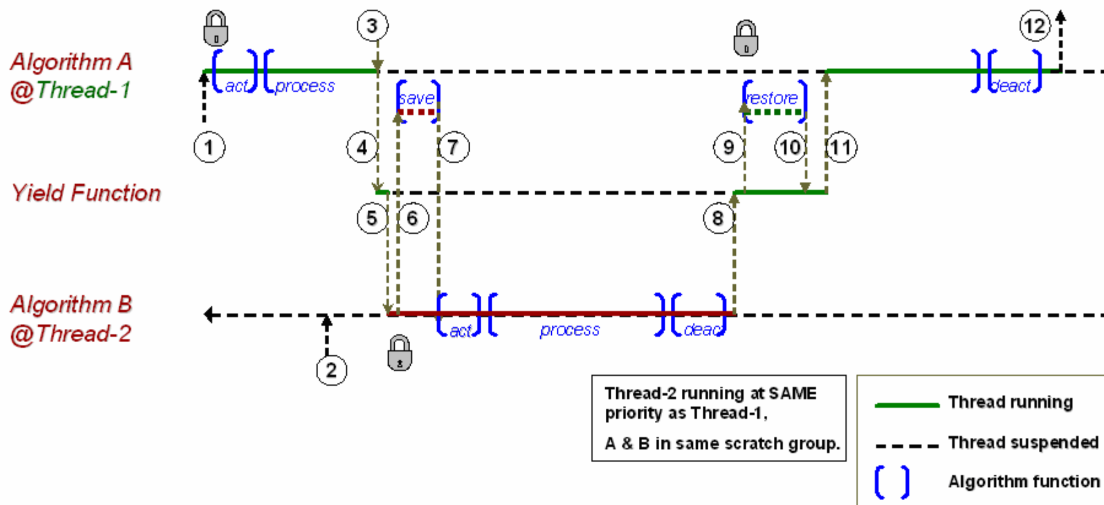

Figure 4. Cooperative Multi-tasking: Yielding to the Same Priority

2.3.3 Cooperative Multi-tasking – Yielding to Higher Priority

Error! Reference source not found. shows the event sequence for a low priority thread that yields to a higher priority thread. The steps are as follows:

1. The framework application starts processing in the context of Thread. Event 1 corresponds to an asynchronous arrival of a request to perform a 'process' operation on algorithm instance, A. In this example Thread 1 is ready to run, so it starts by acquiring a group resource lock, which succeeds. It then activates algorithm instance A, and calls *A's process function.*

2. Framework receives asynchronous request to run Higher Priority Thread 2. An O/S context switch occurs to run Thread 2, however, this thread blocks in attempting to acquire the group-lock, which forces a context switch back to Thread 1.

3. Thread 1, algorithm A, inside its process() function calls IRES Yield.

4. Thread 1, executing the framework supplied Yield() function, releases the group-lock,

5. Thread 2, which was blocked on the group-lock, can now run, and an O/S context switch to Thread 2 takes place.

6. Thread 2 acquires group-lock, checks the 'yield context', and discovers that there is a "yielding" algorithm, A. Therefore it saves A's context by calling the Yield-Context supplied_A::contextSave()
7. Thread 2, returns from saving yielding algorithm A's context, activates algorithm B, calls its process(), which runs to completion. Then it deactivates B.
8. Thread 2 is done, exits or blocks waiting for the next request. This allows the O/S to resume Thread 1.
9. Thread 1, inside IRES Yield function, where it had just released the group resource lock, re-acquires-group-lock successfully. The yield function discovers that a context switch has occurred, so it restores its own algorithm A's context by calling Yield Context supplied A::contextRestore.
10. Thread 1 restores A's context, update its internal yield context state.
11. Thread 1 returns from the IRES Yield() function, resuming back to A's process(), function, to the instruction where it had issued Yield.
12. Thread 1, releases group-lock, exits or blocks.

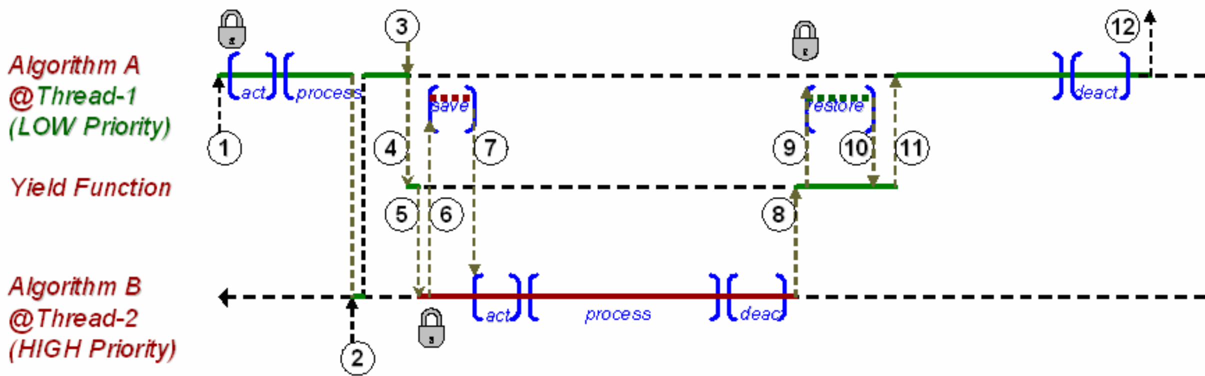

Figure 5. Cooperative Multi-tasking: Yielding to a Higher Priority

3 RMAN: 'C64x+ DMA Resource Manager

3.1 Introduction

RMAN is a generic Resource Manager responsible for granting and reclaiming logical resources to/from requesting algorithms or other software components based on their IRES interface specification. These resources could be, for instance, individual EDMA3 resources (DMA, QDMA channels, PaRams, TCCs) on C64x+ based devices, an HDVICP instance on a Davinci-HD device, a logical IMX unit of the VICP coprocessor on the DM648 device, etc.

The application framework configures RMAN during system start-up (prior to its first use) with appropriate OS-specific memory allocation and semaphore operation functions. At run-time, individual resource managers are registered with RMAN and are then used to grant logical IRES resource handles to algorithms that correspond to physical resources allocated from the system. The granularity of the resource that can be requested depends on the specific implementation of the resource manager, and is described in the IRES interface of the resource (ires_<resource>.h). The resources may be requested and granted as either scratch or persistent resources.

RMAN has the ability to support multiple versions of individual resource managers. It performs compatibility checks between the resource manager (IRESMAN implementation) registered with RMAN, and the version of the IRES interface implemented by the algorithm requesting the resource. A resource is granted only if the two versions are compatible. Other features of RMAN include support for cooperative preemption and multi-tasking. RMAN supports cooperative yield of an algorithm to another of same or higher priority. It uses Framework Components' DSKT2 library to manage contexts of yielding algorithms. For details on this feature see Section 2.3.

RMAN functions are intended to provide application frameworks a convenient and easy-to-use layer to integrate algorithms that request, and in some cases share, hardware resources.

3.2 Using RMAN for Algorithm Integration

The following steps provide a convenient set of instructions for using the RMAN module to grant resources to algorithm instances that request IRES resources. These code snippets are from the RMAN examples included with the Framework Components product. This example demonstrates how RMAN would grant resources to an algorithm that requests EDMA3 resources.

1. Include the RMAN and IRESMAN_EDMA3CHAN module in the application. You can use the RMAN module as provided or make changes to it as needed by the application.

```
#include <ti/sdo/fc/rman/rman.h>
#include <ti/sdo/fc/ires/edma3chan/iresman_edma3Chan.h>
```

2. Call the RMAN module initialization functions and register the resource managers corresponding to the resources that will be requested by the algorithm instances.

```
static IRESMAN_edma3ChanParams configParams;

if (IRES_OK != RMAN_init()) {
        printf("RMAN initialization failed\n");
        return -1;
}

size = sizeof(IRESMAN_Edma3ChanParams);
configParams.baseConfig.allocFxn = RMAN_PARAMS.allocFxn;
configParams.baseConfig.freeFxn = RMAN_PARAMS.freeFxn;
configParams.baseConfig.size = size;

 if (IRES_OK ! = RMAN_register(&IRESMAN_EDMA3CHAN,
         (IRESMAN_Params *)&configParams)) {
    printf("Registration of resource with RMAN failed\n");
}
```

3. Use the RMAN module to grant resources requested by the algorithms.

```
if (IRES_OK != RMAN_assignResources((IALG_Handle) algHandle,
        (IRES_Fxns *)resFxns, scratchGroupId)) {
    printf("Assign Resource Failed \n");
    return -1;
}
```

4. Activate the resources that have been granted to the algorithm *algHandle*.

```
RMAN_activateAllResources((IALG_Handle)algHandle, (IRES_Fxns *)resFxns,
        scratchGroupId);
```

5. Deactivate and then free the resources after the algorithm is done using them. Unregister the resource manager from RMAN and exit if no more algorithms will be assigned resources.

```
RMAN_deactivateAllResources((IALG_Handle)algHandle, (IRES_Fxns *)resFxns,
       scratchGroupId);

if (IRES_OK != RMAN_freeResources((IALG_Handle)(algHandle),
        resFxns, scratchGroupId)) {
    printf("Free Resource Failed \n");
    return -1;
}

if (IRES_OK != RMAN_unregister(&IRESMAN_EDMA3CHAN)) {
        printf("Unregister of Resource Manager Failed \n");
    return -1;
}

RMAN_exit();
```

Note: The above code snippets indicate only the calls to the RMAN module. For more information on how the algorithm requests the IRES resources see Section 2.2.6. For the complete example that includes steps to create algorithm instance (using DSKT2), refer to the RMAN examples that are included as part of the Framework Components product.

3.3 IRESMAN: Standard Interface for Implementing Device-specific Resource Managers

The IRESMAN interface is a standard interface that needs to be implemented by individual resource managers that manage hardware resources in a system. RMAN responds to algorithms' request for resources by obtaining resource handles from these resource managers that have been registered with it.

The IRESMAN interface defined in <ti/sdo/fc/ires/iresman.h> may be extended by individual resource manager implementations to better characterize the actual resource. For instance, the IRESMAN_EDMA3CHAN interface represents and grants EDMA3 resources at the granularity of a single EDMA/QDMA channel, TCC or PaRam set. Several resource manager implementations are available as part of the Framework Components package (EDMA3CHAN, HDVICP, VICP, NULL RESOURCE). The application framework should register the appropriate IRESMAN resource managers with RMAN to be able to satisfy the resource requirements of algorithms being instantiated. Third-party resource manager implementations may also be registered with RMAN as long as they implement the IRESMAN interface.

The following subsections describe the important elements of any IRESMAN resource manager implementation.

3.3.1 IRES Resource Description ires_<resource>.h

This file describes the IRES_<resource>_Obj structure, an extension of the IRES_Obj type defined in ires_common.h, that defines the structure of the logical resource object that is returned to requesting algorithms. It also contains information on what arguments are required to request this resource. It implements and extends the IRES_ProtocolArgs data type to define resource-specific arguments. This file needs to be included by the algorithm that requests the particular IRES resource.

Detailed information on the IRES interface can be found in Sections 2.2.3, 2.2.4 and 2.2.5.

3.3.2 IRESMAN Resource Manager Description iresman_<resource>.h

This file implements the IRESMAN_Fxns interface defined in <ti/sdo/fc/ires/iresman.h> that has APIs used to acquire and release logical IRES handles. RMAN internally calls this interface in response to an algorithm's resource requirements. This file should be included by the application or framework that registers a resource with RMAN.

When the framework registers the IRESMAN resource manager implementation with RMAN using the *RMAN_register* call, it uses two arguments. The first is an implementation of the IRESMAN_Fxns function table, and second, registration parameters that are required to initialize and set up the IRESMAN resource manager.

These two elements are described in detail in the subsections that follow.

3.3.2.1 IRESMAN Registration Parameters (IRESMAN_Params)

IRESMAN_Params are the initialization parameters required by the IRESMAN implementation. These parameters extend the type IRESMAN_Params (ti/sdo/fc/ires/iresman.h) that include memory allocation functions used to allocate IRES resource handles and other internal objects. These parameters may be extended to include specific information that may be used to initialize and configure the device resource manager.

3.3.2.2 IRESMAN Functions (IRESMAN_Fxns)

IRESMAN_Fxns is a global structure to be defined and initialized by individual resource manager implementations. A pointer to this structure must be passed to RMAN during the RMAN_register call. These functions are called by RMAN during various stages of an application. The sections that follow describe the individual functions that need to be implemented and Figure 6 indicates the various stages in the application when the functions are called by RMAN.

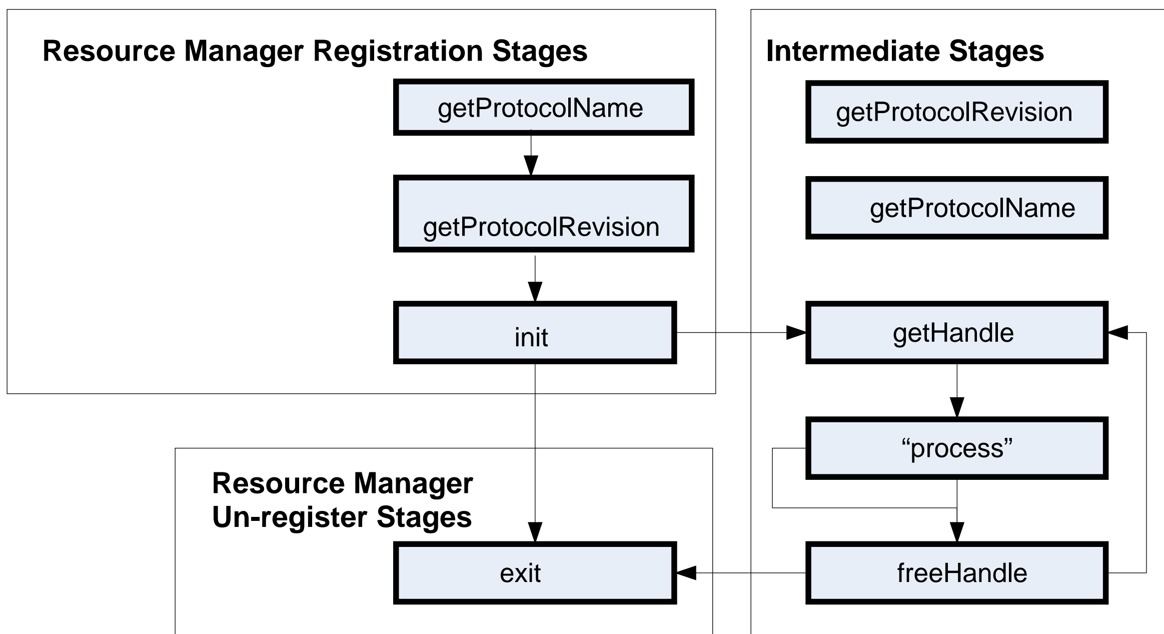

Figure 6. IRESMAN Functions Calling Sequence

3.3.2.3 getProtocolName

This function returns a string representing the name of the protocol. When an algorithm requests a resource, the name of the resource is matched against this string to determine if the resource is available via RMAN. This function is called to identify the IRESMAN resource manager during the initial registration stages, and also during the intermediate stages while serving the requests of an algorithm.

The syntax is:

```
String (*getProtocolName)();
```

3.3.2.4 getProtocolRevision

This function returns a pointer to the IRES_ProtocolRevision type that indicates the version of a particular resource. When an algorithm requests a resource, compatibility between the version of the resource requested and that supported by the IRESMAN resource manager (returned by this call) is checked to determine if the resource can be made available by RMAN. This function is also called to identify the IRESMAN resource manager during the initial registration stages.

The syntax is:

```
IRES_ProtocolRevision * (* getProtocolRevision)();
```

3.3.2.5 init

This function creates and initializes the IRESMAN resource manager implementation. The initialization arguments *IRESMAN_Params* are passed to this function to initialize the resource manager. It is called during the initial registration stages of the resource manager.

The syntax is:

```
IRES_Status (* init)(IRESMAN_Params * initArgs);
```

3.3.2.6 exit

This function deletes the IRESMAN resource manager when it is no longer needed. It is called during the un-registration of the resource manager, in the termination stages of the application framework.

The syntax is:

```
IRES_Status (* exit)();
```

3.3.2.7 getHandle

This function returns resource handles (IRES_Handle) corresponding to the resources requested. The details of the resource request are described in the *IRES_ProtocolArgs* argument passed to this call. The *IRES_ProtocolArgs* type can be extended to include resource specific details. The *IRES_Handle* returned may be a representation of an abstract resource or an actual hardware resource being allocated to a particular algorithm. Like the protocolArgs, the *IRES_Handle* type can be extended to represent the specific resource in detail.

This function is called to grant an algorithm the resources it requires, before it is activated.

The syntax is:

```
IRES_Handle (* getHandle)(IALG_Handle algHandle,
            IRES_ProtocolArgs * protocolArgs, Int scratchGroupId,
            IRES_Status * status);
```

3.3.2.8 freeHandle

This function releases resource handles (IRES_Handle) that are no longer required by an algorithm. The handle is passed to this call that contains the details of the actual resource that is to be freed. This function is called to release the resources held by an algorithm, after it has been de-activated.

The syntax is:

```
IRES_Status (* freeHandle)(IALG_Handle algHandle,
             IRES_Handle resourceHandle, IRES_ProtocolArgs * protocolArgs,
            Int scratchGroupId);
```

3.4 RMAN Configuration

The RMAN module grants abstract IRES resource handles to requesting DSP algorithms. It internally calls the IRES interface of the algorithms to query and grant the IRES resources. It obtains these resources by calling the IRESMAN interface of the individual resource managers that have been registered with it.

Each configuration option can be set at design time by the system integrator to ensure correct operation of RMAN in the execution environment.

There are two ways to configure RMAN parameters:

- You can use a low-level C language based approach to directly modify an interface-defined global configuration structure, RMAN_PARAMS, as defined in the RMAN API specification. The RMAN_PARAMS structure defines the configurable parameters of the RMAN module. Other global variables that describe the internal RMAN table that is used to hold the registered IRESMAN entires can also be modified in a similar manner.
- Alternately, you can use XDC tooling to configure the RTSC module, RMAN. The XDC tooling approach generates the same low-level C based global configuration structures, so the configuration technology used does not matter to the underlying RMAN library implementation.

The following sections describe the configuration parameters of the RMAN module.

3.4.1 XDC Configuration Parameters

The RMAN parameters are configurable using XDC tooling via the RTSC RMAN package interface. These parameters are specified in RMAN.xdc and are listed here.

uInt	RMAN.tableSize
uInt	RMAN.maxAlgs
bool	RMAN.useDSKT2
String	RMAN.persistentAllocFxn
String	RMAN.persistentFreeFxn
bool	RMAN.yieldSamePriority
String	RMAN.semCreateFxn
String	RMAN.semDeleteFxn
String	RMAN.semPendFxn
String	RMAN.semPostFxn
bool	RMAN.debug

These parameters are described in the following list:

- **uInt RMAN.tableSize**

  The total number of entries required in the RMAN_TABLE to hold the various IRESMAN resource managers that will be registered statically or dynamically with RMAN. For example, if RMAN will be managing only the EDMA3 and HDVICP resources, *tableSize* can be set to 2.

- **uInt RMAN.maxAlgs**

  The maximum number of algorithm instances that will use the RMAN module to obtain resources.

- **bool RMAN.useDSKT2**

  A flag indicating if DSKT2 can be used to supply the memory allocation functions and to provide algorithm yield support required by RMAN. If this is set to true, then the following configuration parameter settings are ignored:

  RMAN.persistentAllocFxn,

  RMAN.persistentFreeFxn

  These are internally set to DSKT2's persistent memory allocation and free functions.

- **String RMAN.persistentAllocFxn**

  Function for allocating persistent memory for RMAN's internal objects. This parameter need not be set if *RMAN.useDSKT2* is set to true.

- **String RMAN.persistentFreeFxn**

  Function to free memory allocated by RMAN's *persistentAllocFxn* function. This parameter need not be set if *RMAN.useDSKT2* is set to true.

- **bool RMAN.yieldSamePriority**

  Flag indicating if RMAN should allow yields to algorithms of the same priority when performing cooperative preemption. Since support for cooperative preemption is provided via the DSKT2 library, this parameter is required to be set only if *RMAN.useDSKT2* is true.

- **String RMAN.semCreateFxn**

  Function to create semaphores that might be used by individual resource managers (IRESMAN) registered with RMAN.

- **String RMAN.semDeleteFxn**

  Function to delete semaphores that were created using the RMAN.semCreateFxn function

- **String RMAN.semPendFxn**

  Function to pend on semaphores created using the *RMAN.semCreateFxn* function.

- **String RMAN.semPostFxn**

  Function to post on semaphores created using the *RMAN.semCreateFxn* function.

- **bool RMAN.debug**

  A value of true for the debug parameter enables the debug profile of the RMAN library. This results in a larger and slower version of the library being linked in. It provides extra parameter checking and causes debug statements to be generated in the DSP/BIOS SYS trace buffer.

- **bool RMAN.trace**

  A value of true for the trace parameter enables the trace profile of the RMAN library. This will result in a larger and slower library being linked in, but it will provide trace statements for debugging purposes.

3.4.2 Low-level "C" Configuration Parameters

In a non-XDC environment, configuration parameters may be set or modified using low-level C language. The global RMAN configuration structure (RMAN_PARAMS) and other global variables may be modified to configure RMAN as appropriate.

The following configuration variables and structures are supported:

short	RMAN_Params::numRegistries
IRESMAN_PersistentAllocFxn	* RMAN_PARAMS::allocFxn
IRESMAN_PersistentFreeFxn	* RMAN_PARAMS::freeFxn
IRES_YieldFxn	RMAN_PARAMS::yieldFxn
Bool	RMAN_PARAMS::yieldSamePriority
far IRESMAN_Fxns *	RMAN_TABLE[RMAN_PARAMS_NUMREGISTRIES]
far short	RMAN_FREE_ENTRIES[RMAN_PARAMS_NUMREGISTRIES]
far short	RMAN_numRegistryEntries
far IRESMAN_Fxns	** RMAN_registryEntries
far IRESMAN_Params	** RMAN_registryResmanArgs

These structures and variables are described in this list:

- **short RMAN_Params::numRegistries**

  Number of entries required in the RMAN_TABLE to hold the various IRESMAN resource managers that will be registered statically or dynamically with RMAN. Since a NULL resource is pre-registered with the RMAN_TABLE during initialization, the value of this field should be the total number of resources being registered (statically and dynamically) plus one. See RTSC module RMAN's parameter *RMAN.tableSize*. The value of *numRegistries* is computed by adding one to the configured *tableSize.*

- **IRESMAN_PersistentAllocFxn * RMAN_PARAMS::allocFxn**

  Function for allocating persistent memory for RMAN's internal objects and also to satisfy memory requirements of other IRESMAN implementations. Same as RTSC module RMAN's parameter *RMAN.persistentAllocFxn*.

- **IRESMAN_PersistentFreeFxn * RMAN_PARAMS::freeFxn**

  Function for freeing persistent memory allocated for RMAN's internal objects and other IRESMAN objects. Same as RTSC module RMAN's parameter *RMAN.persistentFreeFxn*.

- **IRES_YieldFxn RMAN_PARAMS::yieldFxn**

  Function to support yields of DSKT2 based algorithms to other algorithms of same or higher priority. If DSKT2 is being used to manage algorithm memory and scratch groups etc, set this to function *RMAN_yield*, else this should be set to NULL.

- **Bool RMAN_PARAMS::yieldSamePriority**

  Flag to indicate if algorithms should yield to others of the same priority. This flag is used only if the *yieldFxn* is set to *RMAN_yield*. Same as RTSC module RMAN's parameter RMAN.yieldSamePriority.

- **far IRESMAN_Fxns * RMAN_TABLE[RMAN_PARAMS_NUMREGISTRIES];**

  Table to store IRESMAN entries that will be registered with RMAN. The size of this table should be the same as *RMAN_PARAMS::numRegistries.* RTSC module RMAN's parameter *RMAN.tableSize* dictates the size of this table.

- **far short RMAN_FREE_ENTRIES[RMAN_PARAMS_NUMREGISTRIES];**

  Table to store the entries that are freed when a resource is un-registered from RMAN using the *RMAN_unregister* API. The size of this table should be the same as *RMAN_PARAMS::numRegistries.* RTSC module RMAN's parameter *RMAN.tableSize* dictates the size of this table.

- **far short RMAN_numRegistryEntries = 0;**

  Number of entries to be registered statically with the Resource Registry table, RMAN. Instead of calling *RMAN_register* to individually register resource managers at run-time, resources could be registered statically by populating these variables *RMAN_numRegistryEntries, RMAN_registryEntries, RMAN_registryResmanArgs*. Populate this variable with the total number of entries that will be registered statically. Information in these variables will be used to pre-register these resource managers during the call to *RMAN_init.*

- **far IRESMAN_Fxns ** RMAN_registryEntries = NULL;**

  Array of IRESMAN (resource manager) configuration entries to be registered statically with RMAN. Instead of calling *RMAN_register* to individually register resource managers at run-time, resources could be registered statically by populating these variables: *RMAN_numRegistryEntries*, *RMAN_registryEntries, RMAN_registryResmanArgs.* Populate this variable with an array of IRESMAN_Fxns pointers that correspond to the resource managers that need to be registered statically. Information in these variables will be used to pre-register these resource managers during the call to *RMAN_init.*

- **far IRESMAN_Params ** RMAN_registryResmanArgs = NULL;**

  Initialization arguments for the IRESMAN configuration entries to be registered statically with RMAN. Instead of calling RMAN_register to individually register resource managers at run-time, resources may be registered statically by populating these variables: RMAN_numRegistryEntries, RMAN_registryEntries, RMAN_registryResmanArgs. Populate this variable with an array of Pointers to initialization arguments for various IRESMAN resource managers. Information in these variables will be used to pre-register these resource managers during the call to RMAN_init.

3.5 RMAN Configuration Examples

The following subsections discuss some common RMAN configuration scenarios.

3.5.1 Static and Dynamic Registration of Resource Managers with RMAN

As mentioned in the previous section, resources can be registered both statically and dynamically with RMAN. The size of the RMAN table needs to be configured large enough to hold all the entries that are registered including the "NULL resource" that is pre-registered with RMAN by default.

If you use XDC to configure RMAN, then you do not need to account for the NULL resource. Set *RMAN.tableSize* to be equal to the total number of entries (static and dynamic) that will be registered with RMAN. Internally the XDC tooling generated "C" code will adjust for the one extra entry that is required.

For non-XDC configuration, set the field *numRegistries* in the *RMAN_PARAMS* structure to a value that is equal to the total number of entries that will be registered with RMAN plus one. The extra entry is required for the NULL resource entry that will be pre-registered with RMAN at init time.

Static registration of resources is currently not available via XDC configuration. To configure static registration of resources, simply modify the following global variables before calling RMAN_init:

- *RMAN_numRegistryEntries* to the number of entries being registered statically.
- *RMAN_registryEntries* to point to an array of pointers to the IRESMAN_Fxns, i.e. the individual resource manager implementation function table.
- *RMAN_registryResmanArgs* to point to an array of pointers to the IRESMAN_Params, i.e. the initialization arguments required by the individual resource manager implementations.

3.5.2 Configuring RMAN to use DSKT2

RMAN can be configured to use Framework Components' DSKT2 library to manage its memory allocation, etc. and also to provide support for cooperative preemption to algorithms.

Using XDC tooling, setting *RMAN.useDSKT2* to *true* lets the DSKT2 library provide appropriate memory allocation functions, and a suitable "yield" function that is required to support cooperative preemption among algorithms. If *.useDSKT2* is set to true, then the following RMAN configuration parameters need not be set individually:

- *RMAN.persistentAllocFxn*
- *RMAN.persistentFreeFxn*

These functions are set to appropriate DSKT2 memory allocation functions. Configured in this manner, RMAN will now allow algorithms to perform cooperative preemption. To control whether an algorithm should be preempted by an algorithm of the same priority, set *RMAN.yieldSamePriority* to true or false.

Even in the absence of XDC tooling, DSKT2 may be used to manage memory allocation and to support cooperative preemption. However, individual fields of the *RMAN_PARAMS* structure need to be set to corresponding DSKT2 functions:

- RMAN_PARAMS.allocFxn should be set to DSKT2_allocPersistent function.
- RMAN_PARAMS.freeFxn should be set to DSKT2_freePersistent function.

If cooperative preemption is to be supported, set *RMAN_PARAMS.yieldFxn* to the ***RMAN_yield*** function exposed by the RMAN module.

Also set the field *yieldSamePriority* to indicate support for yields to algorithms of the same priority.

3.5.3 Configuring RMAN to Not Use DSKT2

RMAN can be configured independently of DSKT2 to use other application- or framework-supplied functions to perform memory allocation.

Using XDC tooling, set *RMAN.persistentAllocFxn* to an appropriate memory allocation function of the type:

```
typedef Bool (fxn)(IALG_MemRec * memTab, Int numRecs)
```

For the *RMAN.persistentFreeFxn*, supply a function of the type:

```
typedef Void (fxn)(IALG_MemRec *memTab, Int numRecs)
```

These functions will be used to allocate and free memory for internal objects required by RMAN.

In the traditional "C" approach, simply set the following parameters of the *RMAN_PARAMS* structure to functions of the type mentioned above: *allocFxn, freeFxn.*

Note: Cooperative preemption support is available only if DSKT2-supplied functions are used to perform memory allocation/free/ Please do not attempt to supply a customized yield function; simply leave the field as NULL. Currently only the DSKT2-based yield function (*RMAN_yield*) is supported, and only when the DSKT2 library is used to manage the scratch groups.

3.6 Other Configuration Parameters

Other configuration parameters for RMAN include functions to create and delete semaphores, and functions to pend/post on them.

RMAN doesn't use these semaphore functions, but some IRESMAN implementations may use RMAN's configured semaphore function implementations to protect access to critical sections in the code or to prevent simultaneous access to global registers/data structures.

Using XDC tooling, configure the following parameters of the RTSC module RMAN:

- *RMAN.semCreateFxn* to a function with the signature *Void * fxn(Int key, Int count);*
- *RMAN.semDeleteFxn* to a function with the signature *Void fxn(Void * sem;)*
- *RMAN.semPendFxn* to a function with signature *Int fxn(Void * sem, unsigned int timeout);*
- *RMAN.semPostFxn* to a function with signature *Void fxn(Void * sem);*

Since these functions are not used by RMAN directly, they are not part of the RMAN_Params structure and are not required to be configured in a non-XDC environment. You would directly configure the semaphore functions explicitly for the modules that require semaphores (for example, IRESMAN_EDMA3CHAN).

3.7 Configuring RMAN Without Using RTSC

The non-RTSC way to configure the RMAN configuration parameters is straightforward. Include the *rman.h* header in your C file and override each of the parameters with the modified values. By default, an RMAN_TABLE and an RMAN_FREE_ENTRIES table will be created of size 10. To change the defaults, create corresponding variables of desired size. See the code snippet below as an example of how to configure RMAN without using RTSC:

```
#include <xdc/std.h>
#include <ti/sdo/fc/rman/rman.h>

#define RMAN_PARAMS_NUMREGISTRIES 5
extern Bool DSKT2_allocPersistent(IALG_MemRec *memTab, Int numRecs);
extern Void DSKT2_freePersistent(IALG_MemRec *memTab, Int numRecs);

/* Table to store the registration entries */
far IRESMAN_Fxns * RMAN_TABLE[RMAN_PARAMS_NUMREGISTRIES];
far short RMAN_FREE_ENTRIES[RMAN_PARAMS_NUMREGISTRIES];

Int main(Void)
{
    RMAN_PARAMS.numRegistries = RMAN_PARAMS_NUMREGISTRIES;
    RMAN_PARAMS.allocFxn                = DSKT2_allocPersistent;
    RMAN_PARAMS.freeFxn                 = DSKT2_freePersistent;
    RMAN_PARAMS.yieldFxn                = RMAN_yield;
    RMAN_PARAMS.yieldSamePrioirty       = true;
    . . .
}
```

3.8 Configuring RMAN Using RTSC Tooling

In the program configuration file, configure the RTSC module RMAN's parameters whose default values you need to change. See the snippet of a *.cfg* file below that configures RMAN:

```
/*
 * Configure RMAN module
 */
var RMAN = xdc.useModule('ti.sdo.fc.rman.RMAN');

RMAN.useDSKT2 = false;
RMAN.persistentAllocFxn = "myAllocFxn" ;
RMAN.persistentFreeFxn = "myFreeFxn" ;
RMAN.tableSize = 10;
RMAN.semCreateFxn = "mySemCreate";
RMAN.semDeleteFxn = "mySemDelete";
RMAN.semPendFxn = "mySemPend";
RMAN.semPostFxn = "mySemPost";
```

3.9 RMAN Version support

RMAN has support for versioning of IRES resources that are requested by the algorithms and whose corresponding resource managers are registered with RMAN. The versions are of the type *IRES_ProtocolRevision* defined in the file *<ti/xdais/ires_common.h>*.

Each resource is associated with a version that is defined in its header file *ires_<resource>.h*

The version has three parts, namely the Major, Source and Radius. RMAN checks all parts of the version returned by the IRESMAN implementation, against that specified by an algorithm requesting a resource of the same name.

Currently RMAN does a check to ensure that the versions are an exact match. It will raise an error when an algorithm is built with a different version of a resource protocol definition than the version used by the registered IRESMAN resource manager implementation.

3.10 RMAN Functions

RMAN functions perform the following resource management functions:

1. Assign and reclaim resources that algorithms request and that are supported by the individual resource managers registered with RMAN.
   - RMAN uses the IRES interface of algorithms to query and grant different resources from/to requesting algorithms.
   - RMAN uses the IRESMAN interface of individual resource managers to obtain handles to different resources as requested by an algorithm.
2. Allocates and frees memory using the memory allocation/free functions that are configured for it.
   - Allocates memory for IRES_ResourceDescriptor objects required to query for resource requirement information from algorithms and to supply that to IRESMAN implementations.
3. Activates and deactivates individual or all resources assigned to a particular algorithm.
   - Resources can be individually or collectively activated after the corresponding algorithms memory has been activated (for example, after DSKT2_activate is called for an algorithm).
   - Resources can be individually or collectively deactivated before the corresponding algorithms' memory has been deactivated (for example, before DSKT2_deactivate is called for an algorithm).

The RMAN API includes several functions that are discussed in detail in the following subsections.

Figure 7 shows the calling sequence of the RMAN API functions.

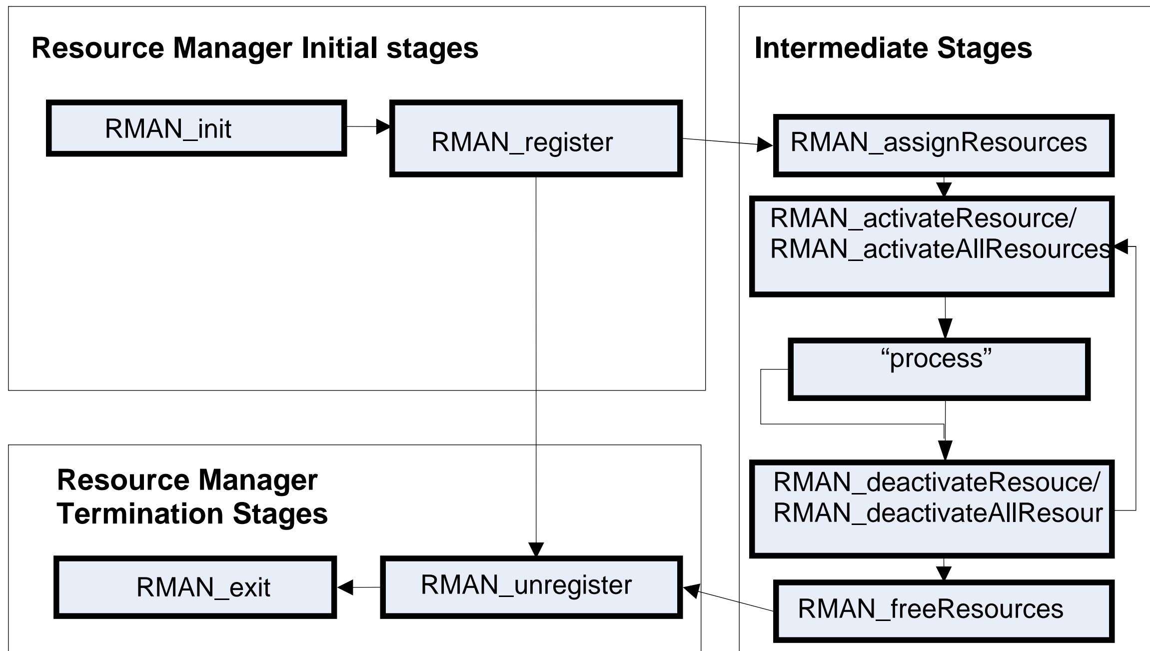

Figure 7. RMAN Functions Calling Sequence

3.10.1 RMAN_exit

This is the finalization method for the RMAN module. There are no parameters and the return value indicates if the call was successful. This function should be called if RMAN will no longer be required by the application or framework.

The syntax is:

```
IRES_Status RMAN_exit(Void);
```

Parameters:

None

Return Values:

IRES_OK	Success.
IRES_EFAIL	Fatal error; un-registering some of the resources failed.

3.10.2 RMAN_init

This is the initialization method for the RMAN module. There are no parameters and the return value indicates if the call was successful. This function needs to be called before any other RMAN function is called.

The syntax is:

```
IRES_Status RMAN_init(Void);
```

Parameters:

None

Return Values:

IRES_OK	Success.
IRES_ENOMEM	Memory allocation failed
IRES_ENORESOURCE	Error registering NULL resource or other statically registered resources.

3.10.3 RMAN_register

This registers an IRESMAN resource manager implementation with RMAN. RMAN stores these entries in an internal resource registry table along with their version information and compares them against algorithms' resource requests.

The syntax is:

```
IRES_Status RMAN_register(IRESMAN_Fxns * resmanFxns,
                          IRESMAN_Params * initArgs);
```

Parameters:

resmanFxns	Resource specific implementation of the IRESMAN interface functions (IRESMAN_Fxns).
initArgs	Arguments to the initialization interface (init) of the IRESMAN implementation being registered.

Return Values:

IRES_OK	Success.
IRES_ENOINIT	RMAN_init call not completed successfully yet.
IRES_EEXISTS	Same protocol is being re-registered.
IRES_EFAIL	Fatal error; exit call on the older entry of the same protocol (same name, same version) failed.
IRES_ENORESOURCE	Initialization of IRESMAN implementation failed.

3.10.4 RMAN_unregister

This un-registers an IRESMAN resource manager implementation when RMAN will no longer need to grant the particular resource to a requesting algorithm.

The syntax is:

```
IRES_Status RMAN_unregister(IRESMAN_Fxns * resmanFxns);
```

Parameters:

resmanFxns	Resource-specific implementation of the IRESMAN interface functions (IRESMAN_Fxns) to be un-registered.

Return Values:

IRES_OK	Success.
IRES_ENOINIT	RMAN_init call not completed successfully yet, or RMAN_exit has already been called.
IRES_ENOTFOUND	Could not find the entry being un-registered in the internal RMAN table.
IRES_EFAIL	Fatal error; exit call on the IRESMAN implementation failed.

3.10.5 RMAN_assignResources

This function assigns one or more resources of the same or different type, requested by an algorithm. It matches algorithm requests with a list of registered IRESMAN resource managers to obtain the resource handles and pass them back to the algorithm.

The syntax is:

```
IRES_Status RMAN_assignResources(IALG_Handle algHandle,
                                 IRES_Fxns * resFxns,
                                 Int scratchGroupId);
```

Parameters:

algHandle	Algorithm handle.
resFxns	IRES interface implementation handle of the algorithm.
scratchGroupId	Scratch group number to which the algorithm belongs and/or from which the resource is being requested.

Return Values:

IRES_OK	Success.
IRES_EALG	Call to one of the IRES interface functions of the algorithm failed.
IRES_ENOMEM	Memory allocation failed.

IRES_ENOTFOUND	Count not find registered IRESMAN implementation corresponding to the resources requested by the algorithm.
IRES_EFAIL	Fatal error; could not find previously registered IRESMAN entry, or had trouble freeing previously allocated handles.
IRES_ENORESOURCE	Could not obtain resource handle from the IRESMAN resource manager.

3.10.5.1 RMAN_freeResources

This function frees IRES resources allocated to the algorithm. The resources are returned to the corresponding IRESMAN resource manager. This API should be called when the resources are no longer required by the algorithm, and after the resources have been deactivated.

The syntax is:

```
IRES_Status RMAN_freeResources(IALG_Handle algHandle,
                               IRES_Fxns *resFxns,
                               Int scratchGroupId);
```

Parameters:

algHandle	Algorithm Handle
resFxns	IRES interface implementation handle of the algorithm.
scratchGroupId	Scratch group number in which the algorithm belongs and/or to which the resource is being returned.

Return Values:

IRES_OK	Success.
IRES_EALG	Calls to one of the IRES interface functions of the algorithm failed.
IRES_ENOMEM	Memory allocation failed.
IRES_ENOTFOUND	Count not find registered IRESMAN implementation corresponding to the resources requested by the algorithm.
IRES_EFAIL	Fatal error; could not find previously registered IRESMAN entry or had trouble freeing previously allocated handles.
IRES_ENORESOURCE	Could not obtain list of allocated resource handles from the IRES interface of the algorithm.

3.10.5.2 RMAN_activateResource

This function activates an IRES resource handle held by the algorithm. Use this function whenever a single resource in a particular scratch group is to be activated. It should be called only after the memory associated with a particular algorithm has been activated (i.e., after DSKT2_activate has been called for that particular algorithm).

The syntax is:

```
IRES_Status RMAN_activateResource(IALG_Handle algHandle,
                                  IRES_Handle resourceHandle,
                                  IRES_Fxns * resFxns,
                                  Int scratchGroupId);
```

Parameters:

algHandle	Algorithm handle.
resourceHandle	Handle to the resource being activated.
resFxns	IRES interface implementation handle of the algorithm.
scratchGroupId	Scratch group number in which the algorithm/resource belongs.

Return Values:

IRES_OK	Success.
IRES_EALG	Calls to one of the IRES interface functions of the algorithm failed.

3.10.5.3 RMAN_deactivateResource

This API deactivates an IRES resource handle held by the algorithm. Use this function to deactivate a single resource in a particular scratch group. Call this function before the memory associated with a particular algorithm has been deactivated (i.e., before DSKT2_deactivate has been called for that particular algorithm).

The syntax is:

```
IRES_Status RMAN_deactivateResource(IALG_Handle algHandle,
                                    IRES_Handle resourceHandle,
                                    IRES_Fxns * resFxns,
                                    Int scratchGroupId);
```

Parameters:

algHandle	Algorithm handle.
resourceHandle	Handle to the resource being deactivated.
resFxns	IRES interface implementation handle of the algorithm.
scratchGroupId	Scratch group number in which the algorithm/resource belongs.

Return Values:

IRES_OK	Success.
IRES_EALG	Calls to one of the IRES interface functions of the algorithm failed.

3.10.5.4 RMAN_activateAllResources

This function activates all IRES resource handles held by the algorithm. Use this function to activate all IRES resources in a particular scratch group. Call this function only after the memory associated with a particular algorithm has been activated (i.e., after DSKT2_activate has been called for that particular algorithm).

```
IRES_Status RMAN_activateAllResources(IALG_Handle algHandle,
                                      IRES_Fxns * resFxns,
                                      Int scratchGroupId);
```

Parameters:

algHandle	Algorithm handle.
resFxns	IRES interface implementation handle of the algorithm.
scratchGroupId	Scratch group number in which the algorithm/resource belongs.

Return Values:

IRES_OK	Success.
IRES_EALG	Calls to one of the IRES interface functions of the algorithm failed.

3.10.5.5 RMAN_deactivateAllResources

This function deactivates all IRES resource handles held by the algorithm. Use this function to deactivate all IRES resources in a particular scratch group. Call this function before the memory associated with a particular algorithm has been deactivated (i.e., before DSKT2_deactivate has been called for that particular algorithm).

The syntax is:

```
IRES_Status RMAN_deactivateResource(IALG_Handle algHandle,
                                    IRES_Handle resourceHandle,
                                    IRES_Fxns * resFxns,
                                    Int scratchGroupId);
```

Parameters:

algHandle	Algorithm handle.
resourceHandle	Handle to the resource being deactivated.
scratchGroupId	Scratch group number to which the algorithm/resource belongs.

Return Values:

IRES_OK	Success.
IRES_EALG	Calls to one of the IRES interface functions of the algorithm failed.

4 References

- *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
- *TMS320 DSP Algorithm Standard API Reference* (SPRU360)
- *TMS320 DSP Algorithm Standard Developer's Guide* (SPRU424)
- *TMS320C6000 Peripherals Reference Guide* (SPRU190)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2008, Texas Instruments Incorporated