TMS320 DSP Algorithm Standard API Reference

User's Guide

Literature Number: SPRU360E
February 2005–Revised February 2007

TEXAS INSTRUMENTS

Contents

List of Figures

List of Tables

Read This First

About This Manual

This document is a companion to the *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) and contains all of the APIs that are defined by the *TMS320 DSP Algorithm Interoperability Standard* (also known as XDAIS) specification.

The TMS320 DSP algorithm standard is part of TI's eXpressDSP technology initiative. Algorithms that comply with the standard are tested and awarded an "expressDSP-compliant" mark upon successful completion of the test.

Intended Audience

This document assumes that you are fluent in the C language, have a good working knowledge of digital signal processing and the requirements of DSP applications, and have had some exposure to the principles and practices of object-oriented programming. This document describes the interfaces between algorithms and the applications that utilize these algorithms. System integrators will see how to incorporate multiple algorithms from separate developers into a complete system. Algorithm writers will be able to determine the methods that must be implemented by eXpressDSP-compliant algorithms and how each method works in a system.

How to Use This Manual

This document contains the following chapters:

- **Chapter 1 - Abstract Algorithm Interfaces**, contains the abstract interfaces that are defined by this specification; all eXpressDSP-compliant algorithms must implement the IALG interface.
- **Chapter 2 - Runtime APIs**, contains runtime APIs for algorithms implementing the IDMA2 and IDMA3 interfaces.
- **Chapter 3 - Supplementary APIs**, describes supplementary module APIs that are available to the clients of XDAIS algorithms but are not part of the core run-time support.
- **Appendix A - Source Code Examples**, contains complete source code examples of eXpressDSP-compliant algorithms.

Each interface defined in this document is presented in a common format. The interface documentation in each chapter is organized as a series of reference pages (first alphabetized by interface name and second by function name) that describes the programming interface for each function. Reference pages are also included that describe the overall capabilities of each interface and appears prior to the functions defined by the interface.

Each function reference page includes the name of the function, number and type of all parameters and return values of the function, a brief description of the function, and all preconditions and postconditions associated with the function. Preconditions are conditions that must be satisfied prior to calling the function. Postconditions are all conditions that the function insures are true when the function returns.

Preconditions must be satisfied by the client while postconditions are ensured by the implementation. Application or framework developers must satisfy the preconditions, whereas developers who implement the interfaces must satisfy the postconditions.

Additional Documents and Resources

The TMS320 DSP Algorithm Standard specification is currently divided between two documents:

1. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
2. *TMS320 DSP Algorithm API Reference* (SPRU360)

The *TMS320 DSP Algorithm Standard Rules and Guidelines* document not only describes all the rules and guidelines that make up the algorithm standard, but contains APIs that are required by the standard and full source examples of standard algorithm components as well.

The following documents contain supplementary information necessary to adhere to the TMS320 DSP Algorithm Standard specification:

- *DSP/BIOS User's Guide*
- *TMS320C54x/C6x/C2x Optimizing C Compiler User's Guide*

In addition to the previously listed documents, complete sources to modules and examples described in this document are included in the TMS320 DSP Developer's Kit. This developer's kit includes additional examples and tools to assist in both the development of XDAIS algorithms and the integration of these algorithms into applications.

Text Conventions

The following conventions are used in this specification:

- Text inside back-quotes ('') represents pseudo-code
- Program source code, function and macro names, parameters, and command line commands are shown in a mono-spaced font.

Algorithm Interfaces

This chapter describes all of the abstract interfaces that are defined by the XDAIS specification that apply to all algorithms.

- IALG - algorithm interface defining a framework-independent interface for the creation of algorithm instance objects.
- IDMA2 - algorithm interface for C64x and C5000 devices defining a uniform way to handle the DMA resource.
- IDMA3 - algorithm interface for C64x+ devices defining a uniform way to handle the DMA resources for the EDMA3 controller.

.

Topic **Page**

1.1 Introduction

All XDAIS algorithms **must** implement the IALG interface. XDAIS algorithms that **want** to utilize the DMA resource must implement the IDMA2 interface and use the ACPY2 interface provided by the client to request DMA services. IDMA2 and ACPY2 APIs replace and deprecate the IDMA and ACPY interfaces that were defined in the earlier revisions of the *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) and *TMS320 DSP Algorithm Standard API Reference* (SPRU360). Algorithms that have already been developed using the deprecated IDMA and ACPY APIs remain eXpressDSP-compliant; however, development of new algorithms should follow the new IDMA2/ACPY2 specification.

XDAIS algorithms designed for the C64x+ EDMA3 controller must implement the IDMA3 interface to publish and acquire DMA resources they will use. The IDMA3 interface is similar to the XDAIS IDMA2 interface in terms of its definition and role, but it exposes some physical EDMA3 resources and is no longer a pure hardware abstraction of generic DMA resources.

Each IDMA3 channel can be optionally associated with a custom IDMA3 protocol; ACPY3 is just one example. This feature allows frameworks to support custom DMA service function libraries with custom initialization and finalization functions. Details of the ACPY3 library provided are available in *Using DMA with Framework Components for C64x+ Application Report* (SPRAAG1).

Modern component programming models support the ability of a single component to implement more than one interface. This allows a single component to be used concurrently by a variety of different clients. In addition to a component's concrete interface (defined by its header) a component can, for example, support a trace interface that allows an in-field diagnostics subsystem to start, stop, and control the acquisition of data showing the component's execution trace. If all traceable components implement a common abstract trace interface, tools and libraries can be written that can uniformly control the generation and display of trace information for all components in a system.

Support for multiple interfaces is often incorporated into the development environment using code wizards, the programming language itself, or both. Since the standard only requires the C language, the ability of a module to support multiple interfaces is awkward.

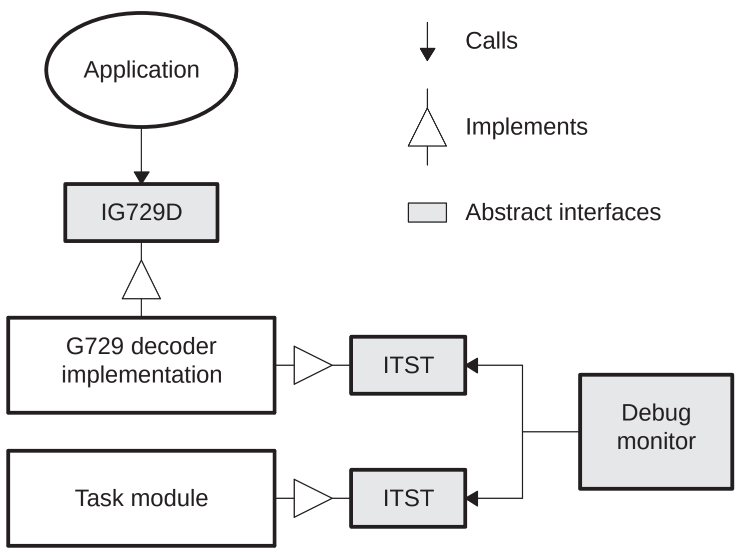


Figure 1-1. Multiple Interface Support

However, several significant benefits make this approach worthwhile. A vendor may opt to not implement certain interfaces for some components. New interfaces can be defined without affecting existing components, and partitioning a large interface into multiple simpler interfaces makes it easier to understand the component as a whole.

As described in the *TMS320 DSP Algorithm Standard Rules and Guidelines* document, interfaces are defined by header files; each header defines a single interface. A module's header file is called a *concrete* interface. A special type of interface header is used to define an *abstract* interface. An abstract interface header is identical to a normal module interface header except that it declares a structure of function pointers named *XYZ_Fxns*. Abstract interfaces are so named because, it is possible that more than one module in a system implements them. A module ABC implements an abstract interface XYZ if it declares and initializes a static structure of type XYZ_Fxns named ABC_XYZ.

1.2 IALG - Algorithm Instance Interface

Synopsis

```
#include <ialg.h>
```

Interface

```
 /*
 *  ======== ialg.h ========
 */
#ifndef IALG_
#define IALG_
#ifdef __cplusplus
extern "C" {
#endif
/*---------------------------*/
/*    TYPES AND CONSTANTS    */
/*---------------------------*/
#define IALG_DEFMEMRECS   4   /* default number of memory records */
#define IALG_OBJMEMREC    0   /* memory record index of instance object */
#define IALG_SYSCMD     256   /* minimum "system" IALG_Cmd value */
#define IALG_EOK          0   /* successful return status code */
#define IALG_EFAI        -1   /* unspecified error return status code */
typedef enum IALG_MemAttrs {
    IALG_SCRATCH,             /* scratch memory */
    IALG_PERSIST,             /* persistent memory */
    IALG_WRITEONCE            /* write-once persistent memory */
} IALG_MemAttrs;
#define IALG_MPROG  0x0008    /* program memory space bit */
#define IALG_MXTRN  0x0010    /* external memory space bit */
/*
 *  ======== IALG_MemSpace ========
 */
typedef enum IALG_MemSpace {
```

```
    IALG_EPROG =     /* external program memory */
    IALG_MPROG | IALG_MXTRN,
    IALG_IPROG =     /* internal program memory */
    IALG_MPROG,
    IALG_ESDATA =    /* off-chip data memory (accessed sequentially) */
    IALG_MXTRN + 0,
    IALG_EXTERNAL =   /* off-chip data memory (accessed randomly) */
    IALG_MXTRN + 1,
    IALG_DARAM0 = 0,  /* dual access on-chip data memory */
    IALG_DARAM1 = 1,  /* block 1, if independent blocks required */
    IALG_SARAM = 2,   /* single access on-chip data memory */
    IALG_SARAM0 = 2,  /* block 0, equivalent to IALG_SARAM */
    IALG_SARAM1 = 3,  /* block 1, if independent blocks required */
    IALG_DARAM2 = 4,  /* block 2, if  3rd independent block required */
    IALG_SARAM2 = 5   /* block 2, if a 3rd independent block required */
} IALG_MemSpace;
/*
 *  ======== IALG_isProg ========
 */
#define IALG_isProg(s) (\
    (((int)(s)) & IALG_MPROG)\
)
/*
 *  ======== IALG_isOffChip ========
 */
#define IALG_isOffChip(s) (\
    (((int)(s)) & IALG_MXTRN) \
)
typedef struct IALG_MemRec {
    Uns     size;         /* size in MAU of allocation */
    Int     alignment;    /* alignment requirement (MAU) */
    IALG_MemSpace space;  /* allocation space */
    IALG_MemAttrs attrs;  /* memory attributes */
```

```
    Void    *base;        /* base address of allocated buf */
} IALG_MemRec;
/*
 *  ======== IALG_Obj ========
 *  Algorithm instance object definition
 *
 *  All XDAS algorithm instance objects *must* have this structure
 *  as their first element. However, they do not need to initialize
 *  it; initialization of this sub-structure is done by the
 *  "framework".
 */
typedef struct IALG_Obj {
    struct IALG_Fxns *fxns;
} IALG_Obj;
/*
 *  ======== IALG_Handle ========
 *  Handle to an algorithm instance object
 */
typedef struct IALG_Obj *IALG_Handle;
/*
 *  ======== IALG_Params ========
 *  Algorithm instance creation parameters
 *
 *  All XDAS algorithm parameter structures *must* have `size'
 *  as their first element.
 */
typedef struct IALG_Params {
    Int size;    /* number of MAU in the structure */
} IALG_Params;
/*
 *  ======== IALG_Status ========
 *  Algorithm specific status structure
 *
```

```
 *  All XDAS algorithm parameter structures *must* have `size'
 *  as their first element.
 */
typedef struct IALG_Status {
    Int size;    /* number of MAU in the structure */
} IALG_Status;
/*
 *  ======== IALG_Cmd ========
 *  Algorithm specific command. This command is used in conjunction
 *  with IALG_Status to get and set algorithm specific attributes
 *  via the algControl method.
 */
typedef unsigned int IALG_Cmd;
/*
 *  ======== IALG_Fxns ========
 *  This structure defines the fields and methods that must be supplied
 *  by all XDAIS algorithms.
 *
 *  implementationId  - unique pointer that identifies the module
 *                      implementing this interface.
 *  algActivate()     - notification to the algorithm that its memory
 *                      is "active" and algorithm processing methods
 *                      may be called. May be NULL; NULL => do nothing.
 *  algAlloc()        - apps call this to query the algorithm about
 *                      its memory requirements. Must be non-NULL.
 *  algControl()      - algorithm specific control operations. May be
 *                      NULL; NULL => no operations supported.
 *  algDeactivate()   - notification that current instance is about to
 *                      be "deactivated". May be NULL; NULL => do
 *                      nothing.
 *  algFree()         - query algorithm for memory to free when
 *                      removing an instance. Must be non-NULL.
 *  algInit()         - apps call this to allow the algorithm to
```

```
 *                       initialize memory requested via algAlloc().
 *                       Must be non-NULL.
 *  algMoved()         - apps call this whenever an algorithms object or
 *                       any pointer parameters are moved in real-time.
 *                       May be NULL; NULL => object can not be moved.
 *  algNumAlloc()      - query algorithm for number of memory requests.
 *                       May be NULL; NULL => number of mem recs is less
 *                       than IALG_DEFMEMRECS.
 */
typedef struct IALG_Fxns {
    Void    *implementationId;
    Void    (*algActivate)(IALG_Handle);
    Int     (*algAlloc)(const IALG_Params *, struct IALG_Fxns **,
              IALG_MemRec*);
    Int     (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
    Void    (*algDeactivate)(IALG_Handle);
    Int     (*algFree)(IALG_Handle, IALG_MemRec *);
    Int     (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle,
              const IALG_Params *);
    Void    (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle,
              const IALG_Params *);
    Int     (*algNumAlloc)(Void);
} IALG_Fxns;
#ifdef __cplusplus

}
#endif
#endif      /* IALG_ */
```

Description

The IALG interface is implemented by all algorithms in order to define their memory resource requirements and enable efficient use of on-chip data memories by client applications.

A module implements the IALG interface if it defines and initializes a global structure of type IALG_Fxns. For the most part, this means that every function defined in this structure must be implemented (and assigned to the appropriate field in this structure). Note that the first field of the IALG_Fxns structure is a Void * pointer. This field must be initialized to a value that uniquely identifies the module implementation. This same value must be used in all interfaces implemented by the module. Since all XDAIS algorithms must implement the IALG interface, it is sufficient for XDAIS algorithm modules to set this field to the address of the module's declared IALG_Fxns structure.

In some cases, an implementation of IALG does not require any processing for a particular method. Rather than require the implementation to implement functions that simply return to the caller, implementations are allowed to set function pointer to NULL. This allows the client to avoid unnecessarily calling functions that do nothing and avoids the code space overhead of these functions.

1. Instance object creation, initialization, and deletion
2. Algorithmic processing
3. Instance object control and relocation

The functions defined in IALG_Fxns fall into several categories. Instance object creation is complicated by removing memory allocation from the algorithm. In order for an algorithm to be used in a variety of applications, decisions about memory overlays and preemption must be made by the client rather than the algorithm. Thus, it is important to give the client as much control over memory management as possible. The functions algAlloc(), algInit(), and algFree( ) allow the algorithm to communicate its memory requirements to the client, let the algorithm initialize the memory allocated by the client, and allow the algorithm to communicate the memory to be freed when an instance is no longer required. Note that these operations are not called in time-critical sections of an application.Please note that the following enhancement affecting allocation, management, and sharing of memory resources is introduced in the current revision of this document (SPRU360C).

- The algorithm may now provide the client, during the algAlloc() call, the base address of any statically initialized IALG_WRITEONCE persistent buffer it is requesting. If the algorithm provides a base address, the client may simply use it to initialize the instance object without allocating any additional memory; otherwise, the client allocates and grants the memory as if it is a standard memory request. However, the client may arrange sharing of write-once persistent buffers by granting multiple instances of the same algorithm created with identical parameters, the same set of write-once persistent buffers. This enhancement provides a simple mechanism for sharing run-time relocatable read-only look-up tables.

Once an algorithm instance object is created, it can be used to process data in real-time. The sub-classes of IALG define other entry points to algorithmic processing supported by eXpressDSP-compliant algorithms. Prior to invoking any of these methods, clients are required to activate the instance object via the algActivate( ) method. The algActivate( ) method provides a notification to the algorithm instance that one or more algorithm processing methods is about to be run zero or more times in succession. After the processing methods have been run, the client calls the algDeactivate method prior to reusing any of the instance's scratch memory. The algActivate( ) and algDeactivate( ) methods give the algorithm a chance to initialize and save scratch memory that is outside the main algorithm-processing loop defined by its extensions of the IALG interface.

The final two methods defined by the IALG interface are algControl( ) and algMoved( ). The algControl( ) operation provides a standard way to control an algorithm instance and receive status information from the algorithm in real-time. The algMoved( ) operation allows the client to move an algorithm instance to physically different memory. Since the algorithm instance object may contain references to the internal buffer that may be moved by the client, the client is required to call the algMoved( ) method whenever the client moves an object instance.

The following figure summarizes the only valid sequences of execution of the IALG_Fxns functions for a particular algorithm instance.

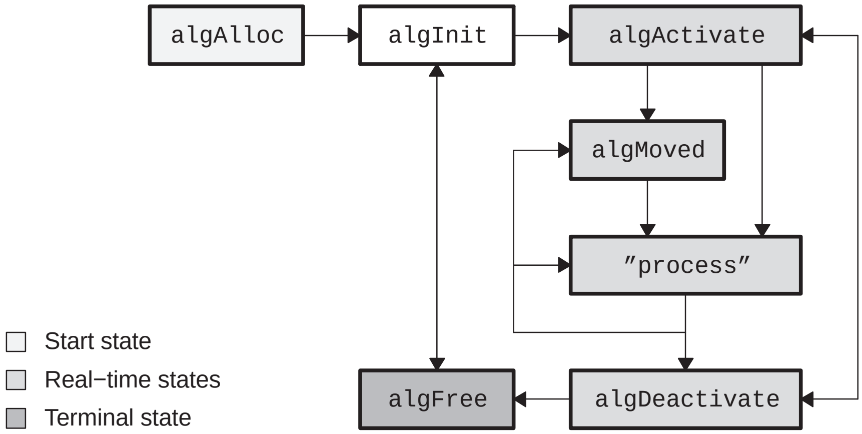


Figure 1-2. IALG Interface Function Call Order

For simplicity, the algControl( ) and algNumAlloc( ) operations are not shown above. The algControl( ) method may be called at any time after algInit( ) and any time before algFree( ). The algNumAlloc( ) method may be called at any time.

Algorithm Parameters and Status

When algorithm instances are created, the client can pass algorithm-specific parameters to the algAlloc( ) and the algInit( ) methods. To support implementation-specific extensions to standard abstract algorithm interfaces, every algorithm's parameter structure must begin with the size field defined in the IALG_Params structure. This field is set by the client to the size of the parameter structure (including the size field itself) that is being passed to the algorithm implementation. Thus, the implementation can "know" if the client is passing just the standard parameter set or an extended parameter set. Conversely, the client can elect to send just the "standard" parameters or an implementation specific set of parameters. Of course, if a client uses an implementation specific set, the client cannot be used with a different implementation of the same algorithm.

```
Client
{
     FIR_Params stdParams;
     FIR_TI_Params tiParams

     stdParams = FIR_PARAMS;               /* initialize all fields to defaults */
     stdParams.coeff = 0;                  /* initialize selected parameters */
     fxns->algAlloc(&stdParams, 0);        /* pass parameters to algorithm */
     tiParams = FIR_TI_PARAMS;             /* initialize all fields to defaults

     tiParams.coeff = 0;                   /* initialize selected parameters
     fxns->algAlloc(&tiParams, 0);         /* pass parameters to algorithm


Int FIR_TI_algAlloc(IALG_Params *clientParams, 0
{
   FIR_TI_Params params = FIR_TI_PARAMS;

    /* client passes in parameters, use them to override defaults
    if (clientParams != NULL)
        memcpy(&params, clientParams, clientParams->size);
    }

    /* use params as the complete set of parameters */
        :

}
```

From the code fragments above, you can see that the client uses the same style of parameter passing when passing generic parameters or implementation-specific parameters. A client may do both. The implementation can also easily deal with either set of parameters. The only requirement is that the generic parameters always form a prefix of the implementation specific parameters; i.e., any implementation specific parameter structure must always include the standard parameters as its first fields.

Example

Algorithms that implement the IALG interface enable run-time instance creation using the following generic create and delete functions.

```
#define MAXMEMRECS 16
typedef struct IALG_Obj {
    IALG_Fxns fxns; /* algorithm functions */
} IALG_Obj;
IALG_Handle ALG_create(IALG_Fxns *fxns, IALG_Params *params)
{
    IALG_MemRec memTab[MAXMEMRECS];
    IALG_Handle alg = NULL;
    Int n;
    IALG_Fxns *fxnsPtr = NULL;
    IALG_Handle p = NULL;

    if (fxns->algNumAlloc() <= MAXMEMRECS) {
        n = fxns->algAlloc(params, &fxnsPtr, memTab);
        if (fxnsPtr != NULL) {
            if ((p = ALG_create(fxnsPtr, NULL)) == NULL)
                return ((IALG_Handle) NULL);
        }
        if (allocMemory(memTab, n)) {
            alg = (IALG_Handle)memTab[0].base;
            alg->fxns = fxns;
            if (fxns->algInit(alg, memTab, p, params) != IALG_EOK) {
                fxns->algFree(alg, memTab);
                freeMemory(memTab, n);
                alg = NULL;
            }
        }
    }
    return (alg);
}
```

1.2.1 algActivate - initialize scratch memory buffers prior to processing

Synopsis

```
Void algActivate(IALG_Handle handle);
```

Arguments

```
IALG_Handle   handle;   /* algorithm instance handle */
```

Return Value

```
Void
```

Description

algActivate( ) initializes any of the instance's scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to algActivate() is an algorithm instance handle. This handle is used by the algorithm to identify the various buffers that must be initialized prior to calling any of the algorithm's processing methods.

The implementation of algActivate() is optional. The algActivate() method should only be implemented if a module wants to factor out initialization code that can be executed once prior to processing multiple consecutive frames of data.

If a module does not implement this method, the algActivate field in the module's static function table (of type IALG_Fxns) must be set to NULL. This is equivalent to the following implementation:

```
Void algActivate(IALG_Handle handle)
{
}
```

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- algActivate( ) can only be called after a successful return from algInit( ).
- handle must be a valid handle for the algorithm's instance object.
- No other algorithm method is currently being run on this instance. This method never preempts any other method on the same instance.
- If the algorithm has implemented the IDMA2 or IDMA3 interface, algActivate( ) can only be called after a successful return from dmaInit( ).

Postconditions

The following condition is true immediately after returning from this method.

- All methods related to the algorithm may now be executed by client (subject to algorithm specific restrictions).

Example

```
typedef struct EncoderObj {
      IALG_Obj ialgObj;          /* IALG object MUST be first field */
      Int *workBuf;              /* pointer to on-chip scratch memory */
      Int *historyBuf;           /* previous frame's data in ext mem */
      ...;
} EncoderObj;

Void algActivate(IALG_Handle handle)
{
      EncoderObj *inst = (EncoderObj *)handle;

      /* copy history to beginning of on-chip working buf */
      memcpy(inst->workingBuf, inst->histBuf, HISTSIZE);
}

Void encode(IALG_Handle handle,
                 Void *in[], Void *out[])
{
      EncoderObj *inst = (EncoderObj *)handle;

      /* append input buffer to history in on-chip workBuf */
      memcpy(inst->workBuf + HISTSIZE, in, HISTSIZE);

      /* encode data */
      ...
      /* move history to beginning of workbuf for next frame */
      memcpy(inst->workBuf, inst->workingBuf + FRAMESIZE, HISTSIZE);
}

Void algDeactivate(IALG_Handle handle)
{
      EncoderObj *inst = (EncoderObj *)handle;

      /* save beginning of on-chip workBuf to history */
      memcpy(inst->histBuf, inst->workingBuf, HISTSIZE);
}
```

See Also

`algDeactivate( )`

1.2.2 algAlloc( ) - get algorithm object's memory requirements

Synopsis

```
numRecs = algAlloc(const IALG_Params *params, IALG_Fxns
                    **parentFxns, IALG_MemRec memTab[]);
```

Arguments

```
IALG_Params *params;    /* algorithm specific attributes */
IALG_Fxns **parentFxns;/* output parent algorithm functions */
IALG_MemRec memTab[]; /* output array of mem records */
```

Return Value

```
Int numRecs; /* number of initialized records in memTab[] */
```

Description

algAlloc() returns a table of memory records that describe the size, alignment, type and memory space of all buffers required by an algorithm (including the algorithm's instance object itself). If successful, this function returns a positive non-zero value indicating the number of records initialized. This function can never initialize more memory records than the number returned by algNumAlloc(). If algNumAlloc() is not implemented, the maximum number of initialized memory records is IALG_DEFMEMRECS.

The first argument to algAlloc() is a pointer to the creation arguments for the instance of the algorithm object to be created. This pointer is algorithm-specific; i.e., it points to a structure that is defined by each particular algorithm. This pointer may be NULL; however, in this case, algAlloc(), must assume default creation parameters and must not fail.

The second argument to algAlloc() is an output parameter. algAlloc() may return a pointer to its parent's IALG functions. If this output value is assigned a non-NULL value, the client must create the parent instance object using the designated IALG functions pointer. The parent instance object must then be passed to algInit().

algAlloc() may be called at any time and it must be idempotent; i.e., it can be called repeatedly without any side effects and always returns the same result.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- The number of memory records in the array memTab[] is no less than the number returned by algNumAlloc().
- *parentFxns is a valid pointer to an IALG_Fxns pointer variable.
- The params parameter may be NULL.

Postconditions

The following conditions are true immediately after returning from this method.

- If the algorithm needs a parent object to be created, the pointer *parentFxns is set to a non-NULL value that points to a valid IALG_Fxns structure, the parent's IALG implementation. Otherwise, this pointer is not set. algAlloc() may elect to ignore the parentFxns pointer altogether.
- For each memory descriptor in memTab with an IALG_WRITEONCE attribute, the algorithm has either set the base field to a non-NULL value, which is the address of a statically allocated and initialized memory buffer of the indicated `size,' or has set the base field to NULL, thereby requiring the memory for the buffer to be provided by the client.
- Exactly n elements of the memTab[ ] array are initialized, where n is the return value from this operation.
- For each memory descriptor in memTab with an IALG_PERSIST or IALG_SCRATCH attribute, the algorithm does not set its base field.
- If the params parameter is NULL, the algorithm assumes default values for all fields defined by the parameter structure.
- memTab[0] defines the memory required for the instance's object and this object's first field is an IALG_Obj structure.
- memTab[0] is requested as persistent memory.

If the operation succeeds, the return value of this operation is greater than or equal to one. Any other return value indicates that the parameters specified by params are invalid.

Example

```
typedef struct EncoderObj {
      IALG_       ialgObj            /* IALG object MUST be first field */
      Int         *workBuf;          /* pointer to on-chip scratch memory */
      Int         workBufLen;        /* expressed in words per frame */
      … ;
} EncoderObj;

typedef struct EncoderParams {
      Int frameDuration;             /* expressed in ms per frame */
};
EncoderParams ENCODERATTRS = {5};    /* default parameters */

Int algAlloc(IALG_Params *algParams, IALG_Fxns **p, IALG_MemRec memTab[])
{
      EncoderParams *params = (EncoderParams *)algParams;

      if (params == NULL) {
          params =       /* use default parameters */
      }

      memTab[0].size = sizeof (EncoderObj);
      memTab[0].alignment = 0;
      memTab[0].type = IALG_PERSIST;
      memTab[0].space = IALG_EXTERNAL;

      memTab[1].size = params->frameDuration * 8 * sizeof(int);
      memTab[1].alignment = 0;       /* no alignment */
      memTab[1].type = IALG_PERSIST;
      memTab[1].space = IALG_DARAM;  /* dual-access on-chip */

      memTab[2].size = sizeof(G729D_VND_EncoderTable);
      memTab[2].alignment = 0;       /* no alignment */
      memTab[2].type = IALG_WRITEONCE;
      memTab[2].space = IALG_SARAM;  /* single-access on-chip */
      memTab[2].base =  /* shared look-up table */


      return (3);
}
```

See Also

`algFree( )`

1.2.3 *algControl - algorithm specific control and status*

Synopsis

```
retval = algControl(IALG_Handle handle,
                    IALG_Cmd cmd, IALG_Status *status);
```

Arguments

```
IALG_Handle  handle;   /* algorithm instance handle */
IALG_Cmd     cmd;      /* algorithm specific command */
IALG_Status  *status;  /* algorithm specific status */
```

Return Value

```
Int retval;
```

Description

algControl( ) sends an algorithm specific command, cmd, and an input/output status buffer pointer to an algorithm's instance object.

The first argument to algControl( ) is an algorithm instance handle. algControl( ) must only be called after a successful call to algInit( ) but may be called prior to algActivate( ). algControl( ) must never be called after a call to algFree( ).

The second and third parameters are algorithm (and possible implementation) specific values. Algorithm and implementation-specific cmd values are always less than IALG_SYSCMD. Greater values are reserved for future upward-compatible versions of the IALG interface.

Upon successful completion of the control operation, algControl( ) returns IALG_EOK; otherwise it returns IALG_EFAIL or an algorithm specific error return value.

In preemptive execution environments, algControl( ) may preempt a module's other methods (for example, its processing methods).

The implementation of algControl( ) is optional. If a module does not implement this method, the algControl field in the module's static function table (of type IALG_Fxns) must be set to NULL. This is equivalent to the following implementation:

```
Int algControl(IALG_Handle handle,
               IALG_Cmd cmd, IALG_Status *status)
       return (IALG_EFAIL);
}
```

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- algControl( ) can only be called after a successful return from algInit( ).
- handle must be a valid handle for the algorithm's instance object.
- Algorithm specific cmd values are always less than IALG_SYSCMD.

Postconditions

The following conditions are true immediately after returning from this method.

- If the control operation is successful, the return value from this operation, retval, is equal to IALG_EOK; otherwise it is equal to either IALG_EFAIL or an algorithm specific return value.
- If the cmd value is not recognized, the return value from this operation, retval, is not equal to IALG_EOK.

Example

```
typedef struct EncoderStatus {
      Bool voicePresent;              /* voice in current frame? */
      … ;
} EncoderStatus;

typedef enum {EncoderGetStatus, 0} EncoderCmd;

Int algControl(IALG_Handle handle,
                 IALG_Cmd cmd, IALG_Status *status)
{
      EncoderStatus *sptr = (EncoderStatus *)status;

      switch ((EncoderCmd)cmd) {
         case EncoderGetStatus:
             sptr->voicePresent = 0;
             …
         case EncoderSetMIPS:
             …
      }
}
```

See Also

algInit( )

1.2.4 algDeactivate - save all persistent data to non-scratch memory

Synopsis

```
Void algDeactivate(IALG_Handle handle);
```

Arguments

```
IALG_Handle  handle;    /* algorithm instance handle */
```

Return Value

```
Void
```

Description

algDeactivate( ) saves any persistent information to non-scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to algDeactivate( ) is an algorithm instance handle. This handle is used by the algorithm to identify the various buffers that must be saved prior to the next cycle of algActivate( ) and processing.

The implementation of algDectivate( ) is optional. The algDeactivate( ) method is only implemented if a module wants to factor out initialization code that can be executed once prior to processing multiple consecutive frames of data.

If a module does not implement this method, the algDectivate field in the module's static function table (of type IALG_Fxns) must be set to NULL. This is equivalent to the following implementation:

```
Void algDeactivate(IALG_Handle handle)
{
}
```

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- algDectivate( ) can only be called after a successful return from algInit( ).
- The instance object is currently "active"; i.e., all instance memory is active and if an algActivate( ) method is defined, it has been called.
- handle must be a valid handle for the algorithm's instance object.
- No other algorithm method is currently being run on this instance. This method never preempts any other method on the same instance.

Postconditions

The following conditions are true immediately after returning from this method.

- No methods related to the algorithm may now be executed by client; only algActivate( ) or algFree( ) may be called.
- All instance scratch memory may be safely overwritten.

See Also

```
algActivate( )
```

1.2.5 algFree - get algorithm object's memory requirements

Synopsis

```
numRecs = algFree(IALG_Handle handle, IALG_MemRec memTab[]);
```

Arguments

```
IALG_Handle handle;   /* algorithm instance handle */
IALG_MemRec memTab[]; /* output array of mem records */
```

Return Value

```
Int numRecs; /* number of initialized records in memTab[] */
```

Description

algFree( ) returns a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm's instance (including the algorithm's instance object itself) specified by handle. This function always returns a positive non-zero value indicating the number of records initialized. This function can never initialize more memory records than the value returned by algNumAlloc( ).

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- The memTab[ ] array contains at least algNumAlloc( ) records.
- handle must be a valid handle for the algorithm's instance object.
- If the prior call to algAlloc( ) returned a non-NULL parent functions pointer, then the parent instance must be an active instance object created via that function pointer.
- No other algorithm method is currently being run on this instance. This method never preempts any other method on the same instance.

Postconditions

The following conditions are true immediately after returning from this method.

- memTab[ ] contains pointers to all of the memory passed to the algorithm via algInit( ).
- The size and alignment fields contain the same values passed to the client via the algAlloc( ) method; i.e., if the client makes changes to the values returned via algAlloc( ) and passes these new values to algInit( ), the algorithm is not responsible for retaining any such changes.

Example

```
typedef struct EncoderObj {
      IALG_Obj      ialgObj /* IALG object MUST be first field */
      Int           *workBuf;
      Int           workBufLen;
      … ;
} EncoderObj;

Int algFree(IALG_Handle handle, IALG_MemRec memTab[])
{
      EncoderObj *inst = (EncoderObj *)handle;

      algAlloc(NULL, memTab);          /* get default values first */

      memTab[0] .size = sizeof (EncoderObj);

      memTab[0] .base = inst;
      memTab[1].size = inst->workBufLen * sizeof(Int);
      memTab[1].base = (Void *)inst->workBuf;
      return(2);
}

Int algAlloc(IALG_Params *params, IALG_MemRec memTab[])
{
      memTab[0].size = sizeof (EncoderObj);
      memTab[0].alignment = 1;
      memTab[0].type = IALG_PERSIST;
```

```
        memTab[0].space = IALG_EXTERNAL;

        memTab[1].size = 80;             /* 10ms @ 8KHz */
        memTab[1].alignment = 1;         /* no alignment */
        memTab[1].type = IALG_PERSIST;
        memTab[1].space = IALG_DARAM;    /* dual-access on-chip */

        return (2);
}
```

See Also

```
algAlloc()
```

In the example code above, algAlloc( ) is called inside algFree( ) to set four out of the five fields in each of the records in memTab[ ]. The purpose of this is to procure some code size optimization by avoiding repetition in algFree( ) of the same code already contained inside algAlloc.

However, careful consideration must be given to this type of optimization since algFree does not take a params argument, while algAlloc does. Some of the fields in a memTab record, typically size, will depend on the params argument. Inside algFree we are forced to call AlgAlloc with NULL (default) params. This value of params may not correspond to the value passed to the original call to algAlloc performed by the client when the algorithm object was instantiated. Because of this, if there are fields in a memTab[ ] record that depend on params, this information must also be stored within the instance object. After algAlloc is called inside algFree, the corresponding fields in the memTab[ ] records should be overwritten to reflect the information stored in the instance object. In the example above, the size field shows this type of behavior.

1.2.6 algInit - initialize an algorithm's instance object

Synopsis

```
status = algInit(IALG_Handle handle, IALG_MemRec memTab[],
                 IALG_Handle parent, IALG_Params *params,);
```

Arguments

```
IALG_Handle handle;   /* algorithm's instance handle */
IALG_memRec memTab[]; /* array of allocated buffers */
IALG_Handle parent;   /* handle algorithm's parent instance */
IALG_Params *params;  /* ptr to algorithm's instance args */
```

Return Value

```
Int  status; /* status indicating success or failure */
```

Description

algInit( ) performs all initialization necessary to complete the run-time creation of an algorithm's instance object. After a successful return from algInit( ), the algorithm's instance object is ready to be used to process data.

The first argument to algInit( ) is an algorithm instance handle. Handle is a pointer to an initialized IALG_Obj structure. Its value is identical to the memTab[0].base.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers allocated for an algorithm instance (including the algorithm's instance object itself. The number of initialized records is identical to the number returned by a prior call to algAlloc( ).

The third argument is a handle to another algorithm instance object. This parameter is often NULL; indicating that no parent object exists. This parameter allows clients to create a shared algorithm instance object and pass it to other algorithm instances. For example, a parent instance object might contain global read-only tables that are used by several instances of a vocoder.

The last argument is a pointer to algorithm-specific parameters that are necessary for the creation and initialization of the instance object. This pointer points to the same parameters passed to the algAlloc( ) operation. However, this pointer may be NULL. In this case, algInit( ), must assume default creation parameters.

The client is not required to satisfy the IALG_MemSpace attribute of the requested memory. Note however that C6000 algorithms that use DMA may strictly require the client to satisfy its on-chip memory requirements and may not function correctly otherwise.

The client may allocate the requested IALG_WRITEONCE buffers once (or never, if the algorithm has assigned a base address in the prior call to algAlloc) and use the same buffers to initialize multiple instances of the same algorithm that are created with identical parameters.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- memTab[ ] contains pointers to non-overlapping buffers with the size and alignment requested via a prior call to algAlloc( ). In addition, the algorithm parameters, params, passed to algAlloc( ) are identical to those passed to this operation.
- handle must be a valid handle for the algorithm's instance object; i.e., handle == memTab[0].base and handle->fxns is initialized to point to the appropriate IALG_Fxns structure.
- If the prior call to algAlloc( ) has not assigned a non-NULL base address to an IALG_WRITEONCE memTab[ ] entry, the client must provide the memory resource with the requested size and alignment.
- If the prior call to algAlloc( ) returned a non-NULL parent functions pointer, then the parent handle, parent, must be a valid handle to an instance object created via that function pointer.
- No other algorithm method is currently being run on this instance. This method never preempts any other method on the same instance.
- If parent is non-NULL, no other method is currently being run on the parent instance; i.e., this method never preempts any other method on the parent instance.
- If the prior call to algAlloc( ) assigned a non-NULL base address to an IALG_WRITEONCE memTab[ ] entry, the client may pass back the same address in the base field without allocating additional memory for that persistent write-once buffer. Alternatively, the client may treat IALG_WRITEONCE buffers in the same way as IALG_PERSIST buffers; by allocating new memory and granting it to the algorithm using the base field.
- The buffer pointed to in memTab[0] is initialized with a pointer to the appropriate IALG_Fxns structure followed by all 0s.

Postconditions

The following conditions are true immediately after returning from this method.

- With the exception of any initialization performed by algActivate( ) and dmaInit( ), all of the instance's persistent and write-once memory is initialized and the object is ready to be used.
- All subsequent memory accesses to the IALG_WRITEONCE buffers by this algorithm instance will be read-only.
- If the algorithm has implemented the IDMA2 interface, the dmaGetChannels( ) operation can be called.

Example

```
typedef struct EncoderObj {
      IALG_Obj ialgObj          /* IALG object MUST be first field */
      int workBuf;              /* pointer to on-chip scratch memory */
      Int workBufLen;           /* workBuf length (in words) */
      … ;
} EncoderObj;

Int algInit(IALG_Handle handle,
             IALG_MemRec memTab[], IALG_Handle p, IALG_Params *algParams)
{
      EncoderObj *inst = (EncoderObj *)handle;
      EncoderParams *params = (EncoderParams *)algParams;

      if (params == NULL) {
         params =     /* use default parameters */
      }

      inst->workBuf = memTab[1].base;
      inst->workBufLen = params->frameDuration * 8;
      …
```

```
    return (IALG_EOK);
}
```

See Also

algAlloc( ), algMoved( )

1.2.7 algMoved - notify instance that instance memory has been relocated

Synopsis

```
Void algMoved(IALG_Handle handle,
              const IALG_MemRec memTab[],IALG_Handle
              parent, const IALG_Params *params);
```

Arguments

```
IALG_Handle handle;   /* algorithm's instance handle */
IALG_Handle parent;   /* handle algorithm's parent instance */
IALG_Params *params;  /*ptr to algorithm's instance args */
IALG_memRec memTab[]; /* array of allocated buffers */
```

Return Value

Void

Description

algMoved() performs any reinitialization necessary to insure that, if an algorithm's instance object has been moved by the client, all internal data references are recomputed.The arguments to algMoved() are identical to the arguments passed to algInit(). In fact, in many cases an algorithm may use the same function defined for algInit() to implement algMoved(). However, it is important to realize that algMoved() is called in real-time whereas algInit() is not. Much of the initialization required in algInit() does not need to occur in algMoved(). The client is responsible for copying the instance's state to the new location and only internal references need to be recomputed.Although the algorithm's parameters are passed to algMoved(), with the exception of pointer values, their values must be identical to the parameters passed to algInit(). The data referenced by any pointers in the params structure must also be identical to the data passed to algInit(). The locations of the values may change but their values must not.The implementation of algMoved() is optional. However, it is highly recommended that this method be implemented. If a module does not implement this method, the algMoved field in the module's static function table (of type IALG_Fxns) must be set to NULL. This is equivalent to asserting that the algorithm's instance objects cannot be moved.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- memTab[ ] contains pointers to all of the memory requested via a prior call to algAlloc( ). The algorithm parameters, params, passed to algInit( ) are identical to those passed to this operation with the exception that pointer parameters may point to different locations, but their contents (what they point to) must be identical to what was passed to algInit( ).
- All buffers pointed to by memTab[ ] contain exact copies of the data contained in the original instance object at the time the object was moved.
- handle must be a valid handle for the algorithm's instance object; i.e., handle == memTab[0].base and handle->fxns is initialized to point to the appropriate IALG_Fxns structure.
- If the prior call to algInit( ) was passed a non-NULL parent handle, then the parent handle, parent, must also be a valid handle to an instance object created with the parent's IALG function pointer.
- algMoved( ) is invoked only when the original instance object is active; i.e., after algActivate( ) and before algDeactivate( ). This allows algMoved to be able to reference both its scratch and persistent memory.
- No other algorithm method is currently being run on this instance. This method never preempts any other method on the same instance.

Postconditions

The following condition is true immediately after returning from this method.

- The instance object is functionally identical to the original instance object. It can be used immediately with any of the algorithm's methods.

Example

```
typedef struct EncoderObj {
      IALG_Obj ialgObj          /* IALG object MUST be first field */
      int workBuf;              /* pointer to on-chip scratch memory */
      Int workBufLen;           /* workBuf length (in words) */
      … ;
} EncoderObj;


algMoved(IALG_Handle handle, IALG_MemRed memTab[],
         IALG_Handle parent, IALG_Params *algParams)
{
      EncoderObj *inst = (EncoderObj *)handle;

      inst->workBuf = memTab[1].base;
}
```

See Also

`algInit( )`

1.2.8 algNumAlloc( ) - number of memory allocation requests required

Synopsis

```
num = algNumAlloc(Void);
```

Arguments

`Void`

Return Value

```
Int num;/* number of allocation requests for algAlloc() */
```

Description

algNumAlloc( ) returns the maximum number of memory allocation requests that the algAlloc( ) method requires. This operation allows clients to allocate sufficient space to call the algAlloc( ) method or fail because insufficient space exists to support the creation of the algorithm's instance object. algNumAlloc( ) may be called at any time and it must be idempotent; i.e., it can be called repeatedly without any side effects and always returns the same result.

algNumAlloc( ) is optional; if it is not implemented, the maximum number of memory records for algAlloc( ) is assumed to be IALG_DEFMEMRECS. This is equivalent to the following implementation:

```
Int algNumAlloc(Void)
{
return (IALG_DEFNUMRECS);
}
```

If a module does not implement this method, the algNumAlloc field in the module's static function table (of type IALG_Fxns) must be set to NULL.

Postconditions

The following condition is true immediately after returning from this method.

- The return value from algNumAlloc( ) is always greater than or equal to one and always equals or exceeds the value returned by algAlloc( ).

Examples

```
#define NUMBUF 3                /* max number of my memory requests */
extern IALG_Fxns *subAlg;       /* sub-algorithm used by this alg */

Int algNumAlloc(Void)
{
      return (NUMBUF + subAlg->algNumAlloc());
```

```
}
Int algAlloc(const IALG_Params *p, struct IALG_Fxns **pFxns,
              IALG_MemRec memTab)
{
      Int n;

      /* initialize my memory requests */
         …
      /* initialize sub-algorithm's requests */
      n = subAlg->algAlloc(0, memTab);
      return (n + NUMBUF);
}
```

See Also

`algAlloc( )`

1.3 IDMA - algorithm DMA interface

This section discusses the IDMA2 interface for C64x and C5000 algorithms and the IDMA3 interface for C64x+ algorithms that use DMA resources.

1.3.1 IDMA2 Interface for C64x and C5000 Algorithms that Use DMA Resources

Synopsis

```
#include <idma2.h>
```

Interface

```
 /*
 *  ======== IDMA2_Handle ========
 *  Handle to "logical" DMA channel.
 */
typedef struct IDMA2_Obj *IDMA2_Handle;

#if defined(_54_) || defined(_55_)
typedef Void (*IDMA2_AdrPtr)();
#define IDMA2_ADRPTR(addr) ((IDMA2_AdrPtr)((LgUns)addr&lt;&lt;1))
#else
typedef Void * IDMA2_AdrPtr;
#endif


/*
*  ======== IDMA2_ElementSize ========
*  8, 16 or 32-bit aligned transfer.
 */
typedef enum IDMA2_ElementSize {
     IDMA2_ELEM8,                /* 8-bit data element  */
     IDMA2_ELEM16,               /* 16-bit data element */
     IDMA2_ELEM32                /* 32-bit data element */
} IDMA2_ElementSize;

/*
*  ======== IDMA2_TransferType ========
*  Type of the DMA transfer.
*/
typedef enum IDMA2_TransferType {
    IDMA2_1D1D,         /* 1-dimensional to 1-dimensional transfer */
    IDMA2_1D2D,         /* 1-dimensional to 2-dimensional transfer */
    IDMA2_2D1D,         /* 2-dimensional to 1-dimensional transfer */
    IDMA2_2D2D          /* 2-dimensional to 2-dimensional transfer */
}   IDMA2_TransferType;
/*
*  ======== IDMA2_Params ========
*  DMA transfer specific parameters. Defines the context of a logical channel.
```

```
*/
typedef struct IDMA_Params {
    IDMA2_TransferType     xType;               /* 1D1D, 1D2D, 2D1D or 2D2D */
    IDMA2_ElementSize   elemSize;               /* Element transfer size */
    Int                UnsFrames;               /* Num of frames for 2D transfers */
    Int    srcElementIndex;   /* In 8-bit bytes */
    Int    dstElementIndex;   /* In 8-bit bytes */
    Int    srcFrameIndex;     /* Jump in 8-bit bytes for 2D transfers */
    Int    dstFrameIndex;     /* Jump in 8-bit bytes for 2D transfers */

} IDMA2_Params;

/*
*  ======== IDMA2_ChannelRec ========
*  DMA record used to describe the logical channels.
*/
typedef struct IDMA2_ChannelRec {
    IDMA2_Handle  handle;    /* Handle to logical DMA channel   */
    Int          queueId;    /* Selects the serialization queue */
} IDMA2_ChannelRec;

/*
*  ======== IDMA2_Fxns ========
*  These fxns are used to query/grant the DMA resources requested by
*  the algorithm at initialization time, and to change these resources
*  at runtime. All these fxns are implemented by the algorithm, and
*  called by the client of the algorithm.
*
*  implementationId     - unique pointer that identifies the module implementing
*                         this interface.
*  dmaChangeChannels()  - apps call this whenever the logical channels are moved
*                         at runtime.
*  dmaGetChannelCnt()   - apps call this to query algorithm about its number of
*                         logical dma channel requests.
*  dmaGetChannels()     - apps call this to query algorithm about its dma channel
*                         requests at init time, or to get the current channel
*                         holdings.
*  dmaInit()            - apps call this to grant dma handle(s) to the algorithm at
*                         initialization.
*/
typedef struct IDMA2_Fxns {
    Void       *implementationId;
    Void      (*dmaChangeChannels)(IALG_Handle, IDMA2_ChannelRec *);
    Int       (*dmaGetChannelCnt)(Void);
    Int       (*dmaGetChannels)(IALG_Handle, IDMA2_ChannelRec *);
    Void      (*dmaInt)(IALG_Handle, IDMA2_ChannelRec *);
} IDMA2_Fxns;

#ifdef ___cplusplus
}
#endif
#endif    /* IDMA2_ */
```

IDMA2 Description

The IDMA2 interface is implemented by algorithms that utilize the DMA resource.

A module implements the IDMA2 interface if it defines and initializes a global structure of type IDMA2_Fxns. Every function defined in this structure must be implemented and assigned to the appropriate field in this structure. Note that the first field of the IDMA2_Fxns structure is a Void * pointer. This field must be initialized to a value that uniquely identifies the module's implementation. This same value must be used in all interfaces implemented by the module. Since all compliant algorithms must implement the IALG interface, it is sufficient for these algorithms to set this field to the address of the module's declared IALG_Fxns structure.

Figure 1-3 illustrates the IDMA functions calling sequence, and also how it relates to the IALG functions.

Note that the dmaChangeChannels( ) and dmaGetChannels( ) operations can be called at any time in the algorithm's real-time stages. The algMoved( ) and algNumAlloc( ) functions are omitted for simplicity.

The important point to notice in the figure above, is that dmaGetChannels( ) and dmaInit( ) operations must be called after algInit( ) and before algActivate( ).

dmaGetChannelCnt( ) can be called before the algorithm instance object is created if the framework wants to query the algorithm of its DMA resource requirements before creating the instance object.

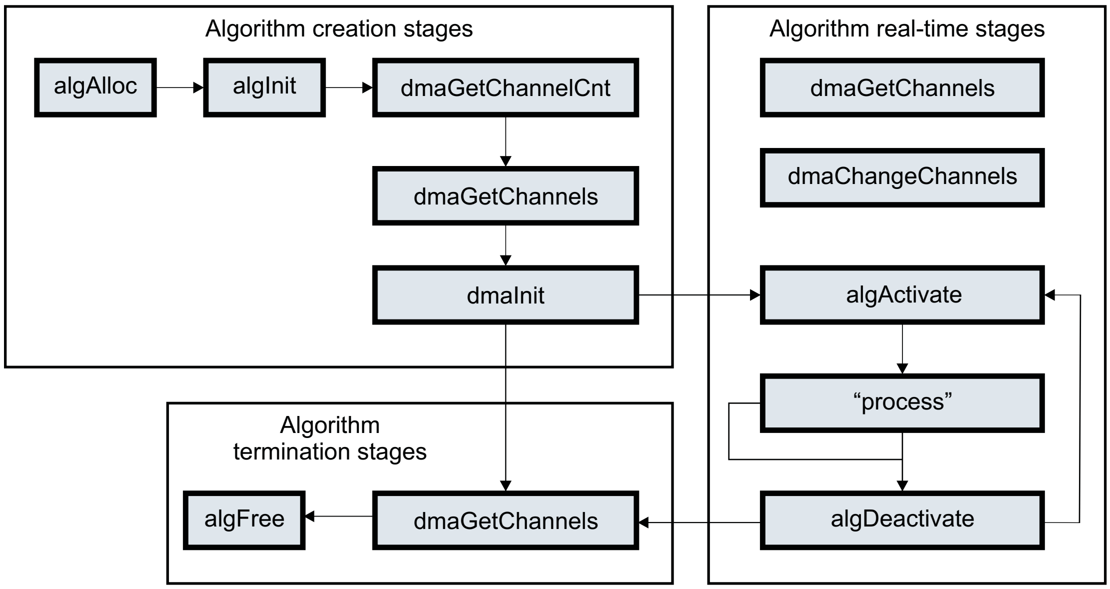


Figure 1-3. IDMA Function Calling Sequence

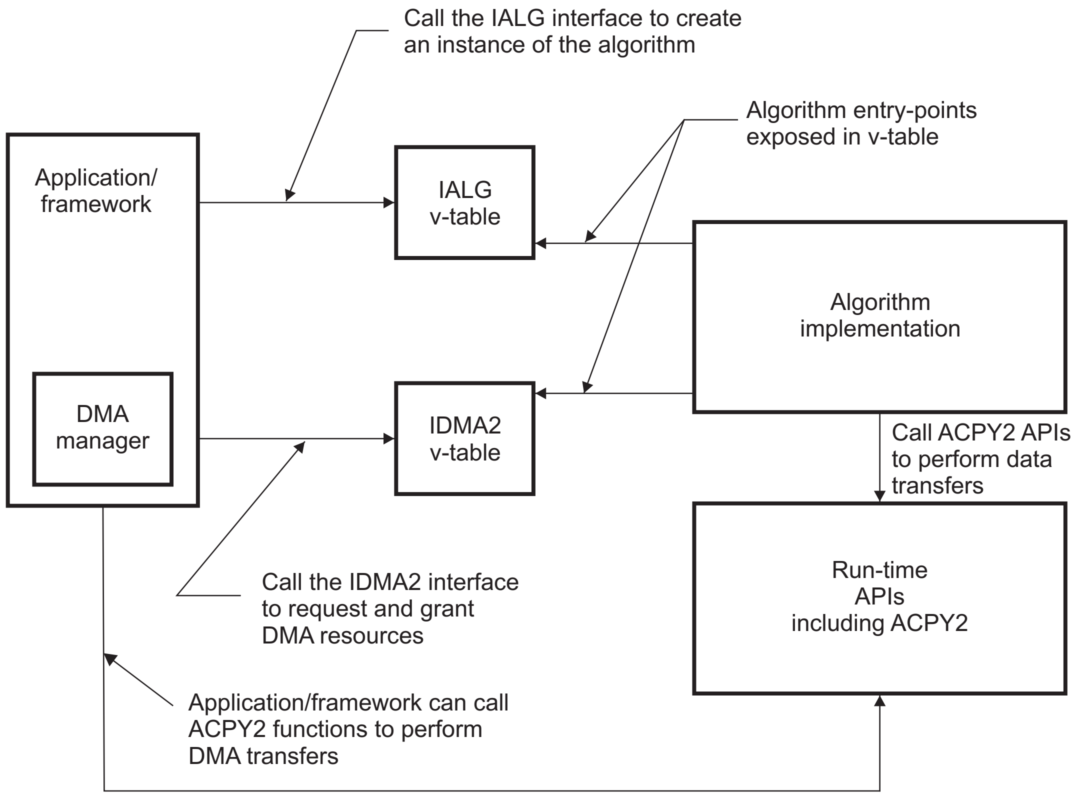


Figure 1-4. Algorithm Implementing the IALG and IDMA2 Interfaces and the Application with a DMA Manage

System Overview

Figure 1-4 illustrates a system with an algorithm implementing the IALG and IDMA2 interfaces and the application with a DMA manager. Notice that the algorithm calls the ACPY2 runtime APIs, which is implemented by the application's DMA manager. The ACPY2 interface describes the comprehensive list of DMA functions an algorithm can call using the DMA handles to program the logical DMA channels obtained through the IDMA2 protocol. These functions allow the algorithm to:

- Configure each logical channel's DMA transfer settings
- Submit asynchronous DMA transfer requests
- Synchronize with completion status of submitted transfers (both blocking and non-blocking)

Logical DMA Channels

Algorithms request DMA services using the hardware abstraction layer described by the IDMA2 and ACPY2 interfaces. Each DMA handle received through the IDMA2_ChannelRec structure provides the algorithm a dedicated, private DMA channel. Each logical DMA channel retains the most recent configuration settings applied by the algorithm and uses those settings when a DMA transfer request is submitted. The physical DMA transfer takes place asynchronously by the available physical hardware under the operational control of the ACPY2 implementation library provided by the client. The same physical DMA resource may be transparently shared among several logical channels owned by one or more algorithm instances.

DMA Transfer Properties

The unit of each DMA transfer is a block made up of frames and elements. Each DMA transfer is scheduled by issuing a source and destination address for the blocks and the number of elements in each frame. The configured properties of the logical channel at the time of transfer request determine the actual memory that gets copied from source to destination. Each DMA transfer is characterized by the following configurable attributes:

- *transfer type* (xType):
  1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
- *element size* (elemSize):
  the number of 8-bit bytes per element {1, 2, 4},
- *number of elements* (arg to ACPY2_start):
  the number of elements per frame, 1 ≤number ≤ 65535
- *number of frames* (numFrames):
  the number of frames in a block, 1 ≤number ≤ 65535
- *element index (srcElementIndex or dstElementIndex)*:
  the size of the gap between two consecutive elements within a frame plus the element size in 8-bit bytes. When element index is set to zero (0), element indexing is not used.
- *frame index (srcFrameIndex or dstFrameIndex)*:
  size of the gap in 8-bit bytes between two consecutive frames within a block. Defined for 2D transfers only.

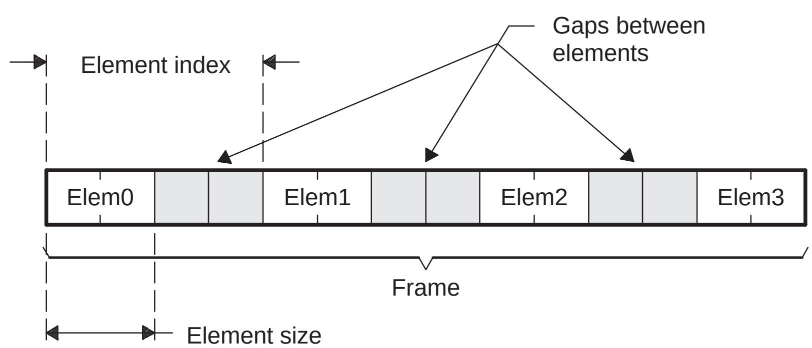


Figure 1-5. Transfer Properties for 1-D Frame

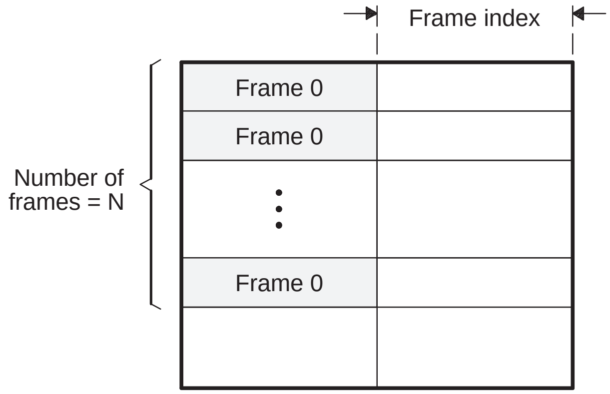


Figure 1-6. Frame Index and 2-D Transfer of –1 Frames

Element and frame index parameters are shared by both source and destination if hardware does not support setting these independently. The ACPY2_configure function should indicate error status when any configuration settings are not supported by the client implementation.

1.3.2 IDMA3 Interface for C64x+ Algorithms that Use the EDMA3 Controller for Data Transfer

Synopsis

```
include <idma3.h>
```

Interface

```
 /**
 *              ti/xdais/idma3.h
 *
 *              IDMA3 Interface Definitions (C64P) - Allows algorithms to
 *              request and receive handles representing private logical DMA
 *              resources.
 */

/**
 *
 *              The IDMA3 interface enables algorithms to request and receive
 *              handles representing private logical DMA resources.
 */
#ifndef IDMA3_
#define IDMA3_


#include "ialg.h"

/**
 *        Memory space descriptors for IDMA3_MemRec.
 */
typedef enum IDMA3_MemType {
    IDMA3_INTERNAL,             /**< Internal data memory. */
    IDMA3_EXTERNAL              /**< External data memory. */
} IDMA3_MemType;


/**
 *              Record containing attributes of the IDMA3 Channel environment
 *              memory when (optionally) the IDMA3 Channel is requested with
 *              a non-NULL IDMA3_ProtocolObj.
 */
typedef struct IDMA3_MemRec {

    /** Base address of allocated memory. */
```

```
    Void *base;

    /** Size of buffer to allocate (MAUs).  */
    Uns size;

    /**
     *  Buffer alignment (0 or a power of 2). Use the values of 0 or 1 to
     *  indicate no alignment requirements.
     */
    Uns align;

    /** Type of memory to use for allocation. */
    IDMA3_MemType memType;

} IDMA3_MemRec;


/**
 *              Handle to "logical" DMA channel.
 */
typedef struct IDMA3_Obj *IDMA3_Handle;


/**
 *              Handle to IDMA3 protocol object
 */
typedef struct IDMA3_ProtocolObj *IDMA3_ProtocolHandle;


/**
 *              NULL protocol definition
 *
 *              The NULL protocol can be used when no protocol is needed,
 *              e.g. when the algorithm directly accesses the eDMA PaRAM
 *              registers, or when the DMA driver library used does not retain
 *              any state. When the NULL protocol is used 'env' buffer is not
 *              allocated
 */
#define IDMA3_PROTOCOL_NULL ((IDMA3_ProtocolHandle)NULL)


/**
 *              IDMA3 Priority Levels.
 */
typedef enum IDMA3_Priority {

    IDMA3_PRIORITY_URGENT,
    IDMA3_PRIORITY_HIGH,
    IDMA3_PRIORITY_MEDIUM,
    IDMA3_PRIORITY_LOW

} IDMA3_Priority;


/**
 *              DMA Channel Descriptor to logical DMA channels.
 */
typedef struct IDMA3_ChannelRec {

    /** Handle to logical DMA channel */
    IDMA3_Handle handle;

    /**
     *  Number of DMA transfers that are submitted using this logical channel
     *  handle. <tt>Single (==1) or Linked ( >= 2).</tt>
     */
```

```
    Int numTransfers;

    /**
     *  Number of individual transfers that can be waited in a linked start.
     *  (1 for single transfers or for waiting all)
     */
    Int numWaits;

    /** Relative priority recommendation: High, Medium, Low. */
    IDMA3_Priority priority;

    /**
     *  When non-NULL, the protocol object provides interface for querying and
     *  initializing logical DMA channel for use by the given protocol. The
     *  protocol can be IDMA3_PROTOCOL_NULL in this case no 'env' is allocated
     *
     *  For example, when requesting a logical channel to be used with ACPY3
     *  APIs, the protocol needs to be set to &ACPY3_PROTOCOL.
     */
    IDMA3_ProtocolHandle protocol;

    /**
     *  When persistent is set to TRUE, the PaRAMs and TCCs will be allocated
     *  exclusively for this channel. They cannot be shared with any other
     *  IDMA3 channel.
     */
    Bool persistent;

} IDMA3_ChannelRec;


/**
 *              These fxns are used to query/grant the DMA resources requested
 *              by the algorithm at initialization time, and to change these
 *              resources at runtime. All these fxns are implemented by the
 *              algorithm, and called by the client of the algorithm.
 */
typedef struct IDMA3_Fxns {

    /**
     *          Unique pointer that identifies the module implementing this
     *          interface.
     */
    Void *implementationId;

    /**
     *          Apps call this whenever the logical channels are moved at
     *          runtime.
     */
    Void (*dmaChangeChannels)(IALG_Handle handle, IDMA3_ChannelRec *chnlRec);

    /**
     *          Apps call this to query algorithm about the maximum number
     *          of logical dma channel requested.
     */
    Uns (*dmaGetChannelCnt)(Void);

    /**
     *          Apps call this to query algorithm about its dma channel
     *          requests at init time, or to get the current channel holdings.
     */
    Uns (*dmaGetChannels)(IALG_Handle handle, IDMA3_ChannelRec *chnlRec);

    /**
     *          Apps call this to grant dma handle(s) to the algorithm at
     *          initialization. Algorithm initializes the instance object.
```

```
     */
    Int (*dmaInit)(IALG_Handle handle, IDMA3_ChannelRec *chnlRec);

} IDMA3_Fxns;


/**
 *              These functions are used to determine memory requirements for
 *              and initialize the IDMA3 protocol's environment that will be
 *              allocated by the DMA manager.
 *
 *              These fxns are implemented by the IDMA3
 *              protocol that is used for a given channel (e.g, ACPY3), and
 *              are passed in the IDMA3_ChannelRec structure to request a
 *              logical DMA channel. If the IDMA3 protocol does not require
 *              any particular one of these functions, it may be set to NULL.
 *
 *              getEnvMemRec() - Called by the DMA manager when creating a
 *              logical DMA channel, to query the IDMA3 protocol's memory
 *              requirements for its environment.
 *
 *              initHandle() - Called by the DMA manager after allocation the
 *              environment, to allow the IDMA3 protocol to do any
 *              initialization of its environment.
 *
 *              deInitHandle() - Called by the DMA manager when a channel is
 *              freed, so that the IDMA3 protocol can do any required
 *              de-initialization or freeing any memory that may have been
 *              allocated in initHandle().
 */
typedef struct IDMA3_ProtocolObj {

    /**
     *  Name of the protocol
     */
    String name;

    /**
     *  Fill in the IDMA3_MemRec with the memory attributes needed for
     *  allocation of the IDMA3 protocol's environment buffer.
     */
    Void (*getEnvMemRec)(IDMA3_ChannelRec *chnlRec, IDMA3_MemRec *memRec);

    /**
     *  Initialize the env stored in the IDMA3 channel handle. Return TRUE on
     *  success, FALSE otherwise. If FALSE is returned channel creation will
     *  fail.
     *
     *  When (and if) the framework/resource manager allocates requested
     *  internal 'env' memory as 'scratch', the 'env' pointer passed in the
     *  IDMA3_Handle points to a persistent, private memory which contains
     *  the address of the 'scratch' allocated 'internal' 'env' memory in the
     *  first word of the 'persistent' 'env' pointer.
     *  If the first word of the env memory is NULL, then no separate 'scratch'
     *  memory has been allocated and 'env' memory itself is 'persistent'.
     */
    Bool (*initHandle)(IDMA3_Handle handle);

    /**
     *  De-Initialize the env stored in the IDMA3 channel handle, before it is
     *  freed by the DMA manager.
     */
    Bool (*deInitHandle)(IDMA3_Handle handle);

} IDMA3_ProtocolObj;
```

```
/**
 *              IDMA3_Obj holds the private state associated with each
 *              logical DMA channel.
 */
typedef struct IDMA3_Obj {

    /** The number of TCCs allocated to this channel. */
    unsigned short numTccs;

    /** The number of PaRam entries allocated to this channel. */
    unsigned short numPaRams;

    /** TCCs assigned to channel */
    unsigned char *tccTable;

    /** PaRAMs assigned to channel */
    Uns *paRamAddr;

    /** Physical QDMA Channel assigned to handle */
    unsigned short qdmaChan;

    /**
     *  Set to true when a new transfer is started on this channel. Set to
     *  false when a wait/sync operation is performed on this channel.
     */
    Bool transferPending;

    /**
     *  IDMA3_ProtocolHandle ('protocol') dependent private channel memory.
     *  The memory for the 'env' is allocated and reclaimed by the framework
     *  when this IDMA3 channel has been requested with a non-NULL 'protocol'.
     *
     *  The size, type and alignment of the allocated 'env' memory is
     *  obtained by calling the channel's 'protocol'->getEnvMemRec() function.
     *
     *  During channel creation, the 'env' pointer must always be created as a
     *  private & persistent memory assigned to the IDMA3 channel object.
     *  However, the framework/resource manager is also allowed to allocate
     *  requested internal 'env' memory as 'scratch' memory which can
     *  only be used when the channel is in active state.
     *  In the 'scratch' allocation case, the framework/resource manager
     *  must still allocate the 'env' as 'persistent',  possibly in
     *  external memory, and must pass the address of the 'scratch' 'internal'
     *  'env' memory in the first word of the 'env' memory.
     *
     *  If the channel 'env' memory is created as 'persistent' with no
     *  'scratch' shadow, then the first word  of the env memory must be
     *  set to NULL.
     */
    Void *env;

    /**
     *  The channel protocol functions used by the DMA manager to determine
     *  memory requirements for the 'env'
     */
    IDMA3_ProtocolHandle protocol;

    /** Indicates if the channel has been allocated with persistent property.*/
    Bool persistent;

} IDMA3_Obj;


#endif  /* IDMA3_ */
```

IDMA3 Description

The IDMA3 interface is implemented by algorithms that need EDMA3 resources. The application framework DMA Resource Manager calls the algorithm's IDMA3 interface functions to query and subsequently allocate and grant the requested DMA resources. (DMAN3 is a reference implementation of a DMA Resource Manager provided by the Framework Components.) If required, the framework calls IDMA3 channel-specific IDMA3_Protocol functions to allocate, initialize, and free additional channel environment memory, which is part of the logical DMA channel state. The algorithm implements the IDMA3 interface by defining and initializing a global structure of type IDMA3_Fxns. Every function defined in this structure must be implemented and assigned to the appropriate field in this structure.

Figure 1-3 illustrates the calling sequence for IDMA3 functions and how these functions relate to the IALG functions performed during algorithm instance creation and real-time operation.

The dmaChangeChannels() and dmaGetChannels() 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 dmaGetChannels() and dmaInit() functions must be called after algInit() and before algActivate(). The dmaGetChannelCnt( ) function can be called before the algorithm instance object is created if the framework wants to query the algorithm about its DMA resource requirements before creating the instance object.

Note: Framework Components provides a DMA resource manager, DMAN3, which provides functions to perform the IDMA3 operations to create algorithms that implement the IDMA3 interface. Details of the Framework Components DMAN3 Resource Manager may be found in *Using DMA with Framework Components for C64x+ Application Report* (SPRAAG1).

Figure 1-7 illustrates a typical system with an algorithm implementing the IALG and IDMA3 interfaces and the application with a DMA manager. Notice that the algorithm calls the ACPY3 run-time functions, which are implemented by the Framework Components. The ACPY3 interface provides a comprehensive list of DMA functions that an algorithm can call using the IDMA3 handles to program the logical DMA channels obtained through the IDMA3 interface. These functions allow the algorithm to:

- Configure each logical channel's DMA transfer settings
- Submit asynchronous DMA transfer requests
- Synchronize with the completion status of submitted transfers (both blocking and non-blocking)

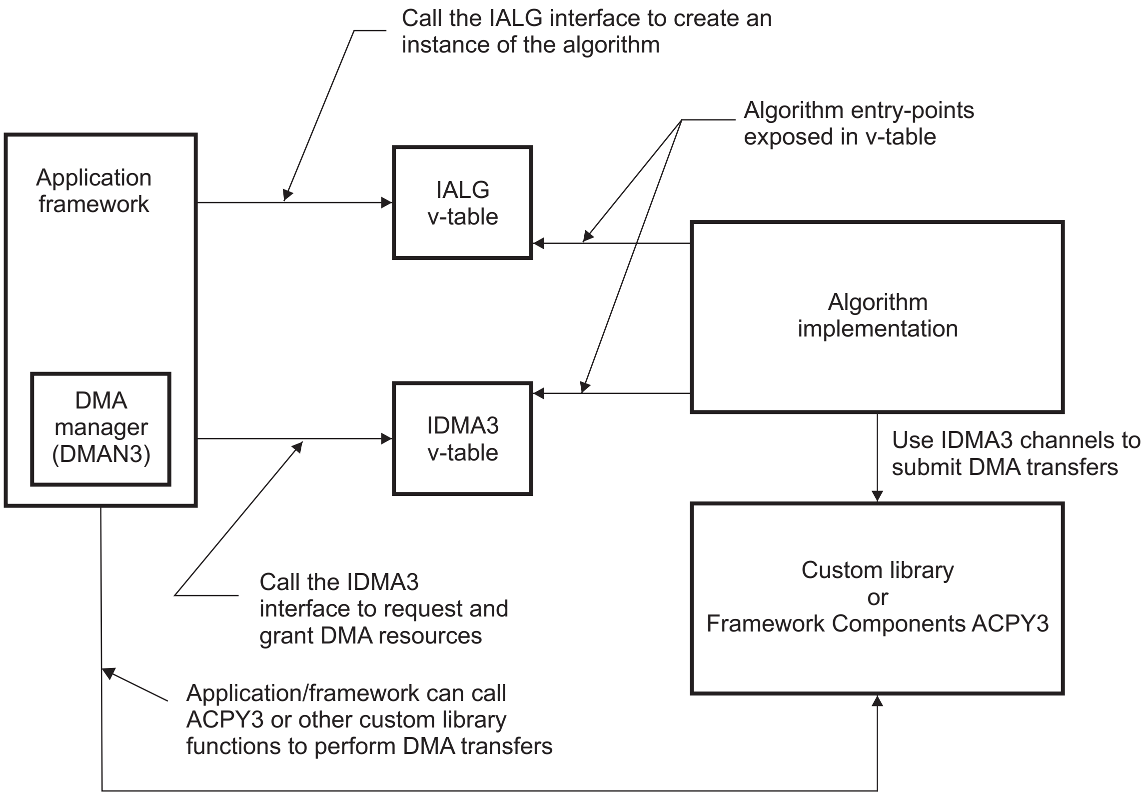


Figure 1-7. Algorithm Implementing IALG and IDMA3 Interfaces and Application Using Framework Components

Logical DMA Channels

The logical DMA channel is the fundamental software abstraction for characterizing hardware DMA resources and services. Each logical DMA channel represents a private hardware DMA resource and a private state identified by and accessed through a DMA handle. Channels obtained through the IDMA3 interface expose the physical EDMA3 resources: Parameter RAM Sets (PaRAMs), Transfer Completion Codes (TCCs), and QDMA Channel ids. Applications are in charge of the physical DMA resources and grant IDMA3 channel handles to algorithms that request them using the IDMA3 interface.

Algorithms or applications can use the physical DMA resources obtained through the IDMA3 interface directly or using any custom DMA library. However, the TI Framework Components package provides a high performance library, ACPY3, which may be used to perform a rich set of DMA operations using the logical DMA channels acquired through the IDMA3 protocol. Details of the ACPY3 library may be found in *Using DMA with Framework Components for C64x+ Application Report* (SPRAAG1).

DMA Transfer Properties

The unit of each DMA transfer is a block made up of frames and elements. Each DMA transfer is scheduled by issuing a source and destination address for the blocks and the number of elements in each frame. The configured properties of the logical channel at the time of transfer request determine the actual memory that gets copied from source to destination. Each DMA transfer is characterized by the following configurable attributes:

- *transfer type* (xType):
  1D-to-1D, 1D-to-2D, 2D-to-1D or 2D-to-2D
- *element size* (elemSize):
  the number of 8-bit bytes per element, 1 ≤ number ≤ 65535
- *number of elements* (arg to ACPY3_start):
  the number of elements per frame, 1≤ number ≤ 65535
- *number of frames* (numFrames):

the number of frames in a block, 1≤ number ≤ 65535

- *element index (srcElementIndex or dstElementIndex)*:

  the size of the gap between two consecutive elements within a frame plus the element size in 8-bit bytes. When element index is set to zero (0) element indexing is not used.
- *frame index (srcFrameIndex or dstFrameIndex)*:

  size of the gap in 8-bit bytes between two consecutive frames within a block. Defined for 2D transfers only.

IDMA3 vs IDMA2

The following key changes have been introduced in IDMA3 interfaces for the C64x+ algorithms as compared to the IDMA2/ACPY2 interfaces for C64x and C5000 algorithms:

- IDMA3 is no longer a pure hardware abstraction of generic DMA resources. Logical channels obtained through the IDMA3 interface expose some physical EDMA3 resources: Parameter RAM Sets (PaRAMs), Transfer Completion Codes (TCCs), and QDMA Channel ids.
- IDMA3 introduces the notion of scratch vs. persistent resources for the physical EDMA3 resources assigned to each IDMA3 channel. This approach is similar to the IALG scratch memory concept, which allows frameworks to efficiently share/overlay algorithm instance scratch buffers using instance activation and deactivation. This approach in IDMA3 allows sharing of TCCs and PaRAM entries and nicely hooks with IALG activate/deactivate events. Channels that cannot be used in a shared context must be requested with “Persistent=TRUE,” otherwise the resource manager is free to arrange the sharing of granted physical DMA resources.
- Each IDMA3 channel can be optionally associated with a custom IDMA3 protocol. When a non-null “protocol” object is provided, the DMA resource manager uses IDMA3_Protocol functions to perform additional memory allocation for the logical DMA channel’s environment (“env”) field or to call protocol-specific handle initialization and de-initialization functions. This feature allows frameworks to support custom DMA service function libraries (ACPY3 is just one such library) with custom initialization and finalization functions.
- The queue IDs defined in IDMA2 are no longer needed. This means there is no requirement to enforce inter-channel FIFO ordering of submitted DMA transfers. When FIFO ordering is needed, you must use linked transfers.
- FIFO completion of DMA transfers is supported only on individual logical DMA channels and linked transfers.
- Support has been added for intermediate synchronization points associated with “wait-ids” for individual transfers within a linked transfer chain

For performance reasons, support was added for the following EDMA3.0-centric DMA concepts:

- Hardware linked transfers that can be quickly started through QDMA.
- Waiting on an intermediate transfer's completion in the case of linked transfers.

IDMA3 Channel Descriptor: IDMA3_ChannelRec

The IDMA3 interface functions use the IDMA3_ChannelRec structure definition to characterize the properties of each logical DMA channel to be granted to the requesting algorithm or module. The DMA Manager utilizes the information passed in a channel descriptor and responds by constructing a logical IDMA3 channel containing the physical DMA resources that are assigned to the channel, and passes the handle of the DMA channel using the same channel descriptor.

IDMA3_Handle	**handle**
Int	numtransfers
Int	numWaits
IDMA3Priority	priority
IDMA3_ProtocolHandle	protocol
Bool	persistent

The following list describes these fields:

- **IDMA3_Handle IDMA3_ChannelRec::handle**
  The handle to a logical DMA channel
- **Int IDMA3_ChannelRec::numTransfers**
  The maximum number of linked DMA transfers that will be submitted using this logical channel handle. Single (==1) or Linked ($\geq$ 2).
- **Int IDMA3_ChannelRec::numWaits**
  The maximum number of transfers that can be independently waited upon. This includes intermediate transfers of a linked DMA transfer. A wait, with a waitId of (numWaits – 1) is configured to indicate the end of the linked or single transfer on a particular channel. Hence, while requesting a handle with a configured number of numWaits, always count the default wait required to indicate the end of transfer. For example, if only 1 intermediate transfer is to be tracked, IDMA3_ChannelRec::numWaits should be 2. Use a waitId of 0 to track the intermediate transfer and a waitId of 1 (numWaits – 1) to track the end of the entire transfer.
- **IDMA3_Priority IDMA3_ChannelRec::priority**
  The relative priority recommendation for transfers submitted on this channel: High, Medium, or Low. See Section 3.2.5 for constants to use for priorities.
- **IDMA3_ProtocolHandle IDMA3_ChannelRec::protocol**
  When non-null, the protocol object provides an interface for querying and initializing logical DMA channel for use by the given protocol. The protocol can be IDMA3_PROTOCOL_NULL; in this case no “env” is allocated. For example, when requesting a logical channel to be used with ACPY3 functions, the protocol needs to be set to &ACPY3_PROTOCOL.
- **Bool IDMA3_ChannelRec::persistent**
  When persistent is set to TRUE, the PaRAMs and TCCs are allocated exclusively for this channel. They cannot be shared with any other IDMA3 channel.

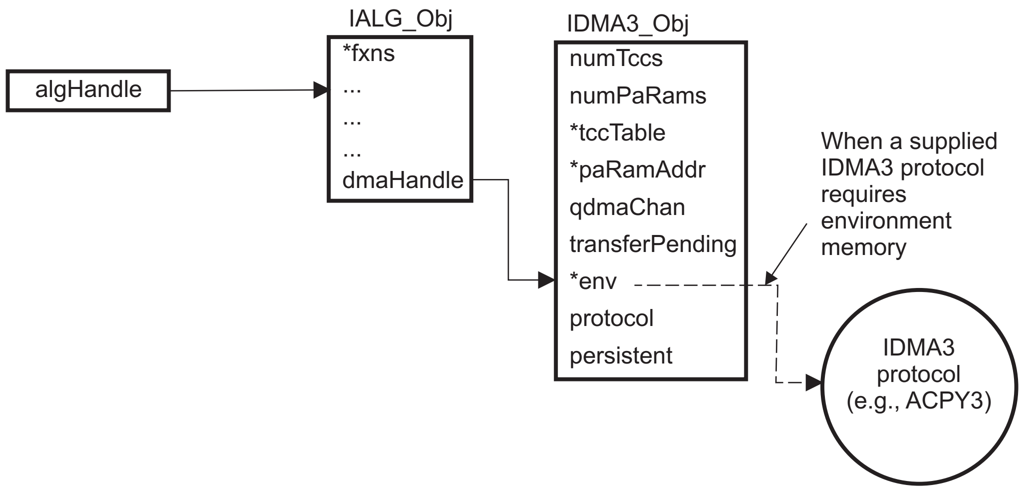
IDMA3 Object and Handle Structures: IDMA3_Obj

The IDMA3 channel object holds the private state associated with each logical DMA channel. The application framework DMA manager creates and initializes the logical channel provisioned with the physical EDMA3 resources that are exposed in this structure definition, and passes its handle to the requesting algorithm using the IDMA3 interface.

The holder of a handle to an IDMA3 channel may directly access the physical resources assigned to the channel or use a standard (ACPY3) or custom DMA functional library that recognizes IDMA3 channel handles.

When the channel is created with its persistent field set to “false,” the physical DMA resources assigned to the channel are be considered to be “scratch” memory, as the definition applies to IALG memory attributes. Algorithms must perform initialization of the resource state each time they are put in an “active” state (via an algActivate call) and must save any necessary channel context when they are deactivated (via algDeactivate). When using ACPY3, calling ACPY3_activate and ACPY3_deactivate during instance activation and deactivation, respectively, performs this required context initialization and deinitialization. Figure 1-8 shows how the algHandle points to an IALG_Obj object, which in turn points to an IDMA3_Obj object.

Figure 1-8. IDMA3 Logical Channels



The IDMA3_Handle data type is a pointer to the IDMA_Obj structure. The IDMA3_Obj structure has the following data fields:

MdUns	numTccs
MdUns	numPaRams
SmUns	tccTable
Uns	paRamAddr
MdUns	qdmaChan
Bool	transferPending
Void	env
IDMA3_ProtocolHandle	protocol
Bool	persistent

The following list describes these fields:

- **MdUns IDMA3_Obj::numTccs**

  The number of TCCs allocated to this channel.
- **MdUns IDMA3_Obj::numPaRams**

  The number of contiguous EDMA3 Parameter RAM (PaRAM Set) register sets allocated to this channel.
- **SmUns* IDMA3_Obj::tccTable**

  The address of the array containing TCCs assigned to this channel.
- **Uns* IDMA3_Obj::paRamAddr**

  The physical address of the first PaRAM assigned to this channel.
- **Bool IDMA3_Obj::transferPending**

  The channel state. This must be maintained by the channel handle owner. The transferPending state must be set to true each time a new DMA transfer is physically submitted using this DMA channel. The state must be cleared to false before a new physical transfer can be submitted using this channel.
- **Void* IDMA3_Obj::env**

  An optional "environment" memory that may be allocated as a private extension of the channel state. Memory for the "env" must be supplied by the framework prior to calling the IDMA3_Fxns::dmaInit function only if the IDMA3 channel descriptor requires it by providing a non-null "protocol" handle and a non-null getEnvMemRec() function pointer.

  The IDMA3_Protocol object's getEnvMemRec() function characterizes the size, alignment and space attributes of the "env" memory needed for the channel. The framework is responsible for allocating and reclaiming the "env" memory.

  During channel creation, the "env" pointer must always be created as private and persistent memory assigned to the IDMA3 channel object. However, the framework/resource manager is also allowed to allocate the requested internal "env" memory as "scratch" memory that can only be used when the channel is in an active state.

In the scratch allocation case, the framework/resource manager must still allocate the "env" as a "persistent" shadow memory, possibly in "external memory", and then pass the address of the scratch internal "env" memory in the first word of the returned IDMA3_Obj's "env" pointer. If a channel's "env" memory is created as "persistent" with no "scratch" shadow, then the first word of the env memory must be set to null.

- **IDMA3_ProtocolHandle IDMA3_Obj::protocol**

  When non-null, this points to the channel protocol function table used by the DMA manager to interrogate and provision memory for the channel’s “env” area.
- **Bool IDMA3_Obj::persistent**

  This flag indicates whether the channel was allocated with the persistent property.

IDMA3 Protocol Object for Channel Environment Memory Management

The IDMA3 protocol object (the protocol field of the channel descriptor) is used only when the requestor of the IDMA3 channel requires:

- additional environment memory (assigned to the channel’s “env” pointer) to be allocated by the framework as part of the channel object, or
- custom initialization or de-initialization functions to be called by the framework upon channel creation and deletion

If these requirements do not apply, you may request the IDMA3 channel with a null "protocol" field and skip the details of this section.

Algorithms or framework libraries (such as ACPY3) may need some additional persistent and private "environment" memory to be associated with each IDMA3 channel in order to manage channel state or to create shadow copies of registers, data structures, etc. The IDMA3 interface defines a protocol, via the IDMA3_ProtocolObj specification, that can be implemented and used by individual algorithms or functional DMA libraries to request and receive channel environment memory from the resource management framework during channel creation.

Custom DMA libraries may take advantage of this feature and require that IDMA3 channels are requested and created using specific IDMA3 Protocol Objects. An example of this is the ACPY3 library, which supplies its own protocol object, ACPY3_PROTOCOL. Algorithms or applications that will use ACPY3 to submit DMA transfers are simply required to set the "protocol" field to the address of ACPY3_PROTOCOL when they request an IDMA3 channel for ACPY3 use.

When the IDMA3 channel descriptor (IDMA3_ChannelRec) contains a non-null IDMA3_ProtocolObj reference, the provided functions are called to determine channel environment memory requirements and to perform initialization and de-initialization of the channel object. If the IDMA3 protocol does not require the functionality associated with any particular function, it may be set to null. The application framework and DMA resource manager are responsible for calling the IDMA3_Protocol functions.

The IDMA3_ProtocolObj contains the following fields:

```
String name
Void (*getEnvMemRec )(IDMA3_ChannelRec *, IDMA3_MemRec *)
Bool (*initHandle )(IDMA3_Handle)
Bool (*deInitHandle )(IDMA3_Handle)
```

The following list describes these fields:

- **String IDMA3_ProtocolObj::name**

  The name of the protocol.
- **Void(*IDMA3_ProtocolObj::getEnvMemRec)(IDMA3_ChannelRec *, IDMA3_MemRec *)**

  The function is called by the application framework/DMA manager to obtain the IDMA3 protocol's memory requirements (IDMA3_MemRec) for its environment for the given IDMA3 channel descriptor. This is usually done when creating a logical DMA channel.
- **Bool(* IDMA3_ProtocolObj::initHandle)(IDMA3_Handle)**

  This function is called after allocation. It allows the IDMA3 protocol to do any initialization of its environment. It initializes the "env" memory passed in the IDMA3 channel handle and any other channel state. Returns TRUE on success, FALSE otherwise. If FALSE is returned channel creation fails. If the framework/resource manager allocates the requested internal "env" memory as "scratch", the "env" pointer passed in the IDMA3_Handle points to a persistent and private shadow memory (possibly in "external memory"), which contains the address of the "scratch" allocated "internal" "env" memory in the first word of the "persistent" "env" pointer. If the first word of the env memory is NULL, then no separate "scratch" memory has been allocated and the "env" memory itself is "persistent."

- **Bool(* IDMA3_ProtocolObj::deInitHandle)(IDMA3_Handle)**
  This function is called when a channel is freed. It deinitializes a channel before assigned resources and memory are freed by the DMA manager. It is called so that the IDMA3 protocol can do any required de-initialization or freeing any memory that may have been allocated in initHandle().

IDMA3 enumeration Type Documentation

The following enumerated types are defined for use by the IDMA3 API:

- **enum IDMA3_MemType**

Constant	Memory Type
IDMA3_INTERNAL	Internal data memory
IDMA3_EXTERNAL	External data memory

- **enum IDMA3_Priority**

Constant	IDMA3 Priority Level
IDMA3_PRIORITY_URGENT	Urgent
IDMA3_PRIORITY_HIGH	High
IDMA3_PRIORITY_MEDIUM	Medium
IDMA3_PRIORITY_LOW	Low

1.3.3 dmaChangeChannels - notify algorithm instance that DMA resources have changed

Synopsis

For IDMA2 interface

```
dmaChangeChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[]);
```

For IDMA3 interface

```
dmaChangeChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]);
```

Arguments

For IDMA2 interface

```
IALG_Handle handle;        /* handle to algorithm instance */
IDMA2_ChannelRec dmaTab[]; /* input array of dma records */
```

For IDMA3 interface

```
IALG_Handle handle;        /* handle to algorithm instance */
IDMA3_ChannelRec dmaTab[]; /* input array of dma records */
```

Return Values

```
Void
```

Description

dmaChangeChannels( ) performs any re-initialization necessary to insure that, if the algorithm's logical DMA channels have been changed by the client, all internal references are updated.

The arguments to dmaChangeChannels() are identical to the arguments passed to dmaInit( ). In fact, in many cases an algorithm may use the same function defined for dmaInit( ) to implement dmaChangeChannels( ). However, it is important to realize that dmaChangeChannels() is called in real-time whereas dmaInit( ) is not.

The first argument to dmaChangeChannels( ) is the algorithm's instance handle.

The second argument to dmaChangeChannels() is a table of dma records that describe the DMA resource. The handle field in the structure must be initialized to contain a value that indicates the new logical DMA channel.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer for the algorithm's instance object.
- The handle field in the dmaTab[ ] array must contain a value indicating the new logical DMA channel.

Postconditions

The following conditions are true immediately after returning from this method.

- The instance object is functionally identical to the original instance object.

Example

For IDMA2

```
typedef struct ImageObj {
    IALG_Obj ialg; /* IALG object MUST be first field */
    IDMA2_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
    IDMA2_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
    IDMA2_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
    Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;
Void dmaChangeChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[]){
    ImageObj *img = (ImageObj *)handle;
    img->dmaHandle_0 = dmaTab[0].handle;
    if (!img->grayScale) {
        img->dmaHandle_1 = dmaTab[1].handle;
        img->dmaHandle_2 = dmaTab[2].handle;
    }
}
```

For IDMA3

```
typedef struct ImageObj {
    IALG_Obj ialg; /* IALG object MUST be first field */
    IDMA3_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
    IDMA3_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
    IDMA3_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
    Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;

Void dmaChangeChannels(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]){
    ImageObj *img = (ImageObj *)handle;
    img->dmaHandle_0 = dmaTab[0].handle;
    if (!img->grayScale) {
        img->dmaHandle_1 = dmaTab[1].handle;
        img->dmaHandle_2 = dmaTab[2].handle;
    }
}
```

See Also

`dmaInit()`

1.3.4 dmaGetChannelCnt - get number of DMA resources required

Synopsis

```
numRecs = dmaGetChannelCnt(Void);
```

Arguments

`Void`

Return Values

```
Int numRecs;  /*number of allocation requests for dmaGetChannels*/
```

Description

dmaGetChannelCnt() returns the maximum number of logical DMA channels requested by each algorithm instance objects for the module. This operation allows the client to allocate sufficient space to call the dmaGetChannels() operation, or fail because of insufficient resources.

Note: dmaGetChannelCnt ( ) can be called before the algorithm instance object is created.

Postconditions

The following conditions are true immediately after returning from this method.

- The return value from dmaGetChannelCnt( ) is always greater than or equal to zero and always equals or exceeds the value returned by dmaGetChannels( ).

Example

For IDMA2

```
typedef struct ImageObj {
    IALG_Obj ialg; /* IALG object MUST be first field */
    IDMA2_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
    IDMA2_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
    IDMA2_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
    Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;

Int dmaGetChannelCnt(Void){
    return(3) /* Maximum 3 logical channels */
}
```

For IDMA3

```
typedef struct ImageObj {
        IALG_Obj ialg; /* IALG object MUST be first field */
        IDMA3_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
        IDMA3_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
        IDMA3_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;

Int dmaGetChannelCnt(Void){
        return(3) /* Maximum 3 logical channels */
}
```

See Also

`dmaGetChannels()`

1.3.5 dmaGetChannels - get algorithm object's dma requirements/holdings

Synopsis

For IDMA2 interface

```
numRecs = dmaGetChannels(IALG_Handle handle,IDMA2_ChannelRec dmaTab[]);
```

For IDMA3 interface

```
numRecs = dmaGetChannels(IALG_Handle handle,IDMA3_ChannelRec dmaTab[]);
```

Arguments

For IDMA2 interface

```
IALG_Handle handle;             /* handle to algorithm instance
IDMA2_ChannelRec dmaTab[];      /* output array of dma records  */
```

For IDMA3 interface

```
IALG_Handle handle;             /* handle to algorithm instance
IDMA3_ChannelRec dmaTab[];      /* output array of dma records  */
```

Return Values

```
Int numRecs;   /* Number of initialized records in dmaTab[] */
```

Description

dmaGetChannels( ) returns a table of dma records that describe the algorithm's DMA resources. The handle field returned in the IDMA2_ChannelRec structure (or IDMA3_ChannelRec for IDMA3 interfaces) is undefined when this operation is called at algorithm's initialization. All fields in this structure are valid if this operation is called after a successful call to the dmaInit( ) operation.

The first argument to dmaGetChannels( ) is the algorithm instance handle. The second argument to dmaGetChannels( ) is a table of dma records that describe the algorithm's DMA resources.

It is important to notice that the number of logical DMA channels that are being requested might be dependent on the algorithm's interface creation parameters. The algorithm is already initialized with these parameters through algInit( ).

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- The number of dma records in the dmaTab[] array is equal to or less than the number returned by dmaGetChannelCnt( ).
- handle must be a valid pointer to the algorithm's instance object.
- dmaGetChannels( ) can only be called after a successful return from algInit( ).

Postconditions

The following conditions are true immediately after returning from this method.

- Exactly numRecs elements of the dmaTab[ ] array are initialized, where numRecs is the return value from this operation.
- The handle field in the IDMA2_ChannelRec (or IDMA3_ChannelRec) structure is valid only if this operation is called after algInit( ).

Example

For IDMA2

```
 typedef struct ImageObj {
        IALG_Obj ialg; /* IALG object MUST be first field */
        IDMA2_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
        IDMA2_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
        IDMA2_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;
typedef struct IMG_Params {
        Int size; /* size of this structure */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} IMG_Params;
const IMG_Params IMG_PARAMS = { /* default parameters */
        sizeof(IMG_PARAMS),
        TRUE
};
Int algInit(IALG_Handle handle,IALG_MemRec memTab[], IALG_Handle p, IALG_Params *algParams)
{
        ImageObj *img = (ImageObj *)handle;
        IMG_Params *params = (IMG_Params *)algParams;
        if (params == NULL) {
              params = &IMG_PARAMS; /* Use interface default parameters */
        }
        /* Initialize the logical DMA channel */
        img->dmaHandle_0 = NULL;
        img->dmaHandle_1 = NULL;
        img->dmaHandle_2 = NULL;
        img->grayScale = params->grayScale;
        return (IALG_EOK);
}
```

```
Int dmaGetChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
        ImageObj *img = (ImageObj *)handle;
        /* algInit() is called before this fxn, so the `grayScale' field */
        /* in the instance object is initialized. */
        return (myDmaTabInit(&img, &dmaTab));
}
static Int myDmaTabInit(ImageObj *img, IDMA2_ChannelRec dmaTab[])
{
        dmaTab[0].handle = img->dmaHandle_0;
        /* If the image is grayscale, require just one logical DMA channel, */
        /* otherwise request three logical channels; one for each color plane */
        if (!img->grayScale) {
               dmaTab[1].handle = img->dmaHandle_1;
               dmaTab[2].handle = img->dmaHandle_2;
               /*
               * Transfers on each logical channel are independent.
               * i.e., each logical channel can be assigned to separate hardware queue.
               */
               dmaTab[0].queueId = 0;
               dmaTab[1].queueId = 1;
               dmaTab[2].queueId = 2;
               return (3);
        }
        return (1)
}
```

For IDMA3

```
 typedef struct ImageObj {
        IALG_Obj ialg; /* IALG object MUST be first field */
        IDMA3_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
        IDMA3_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
        IDMA3_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;
typedef struct IMG_Params {
        Int size; /* size of this structure */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} IMG_Params;
const IMG_Params IMG_PARAMS = { /* default parameters */
        sizeof(IMG_PARAMS),
        TRUE
};
Int algInit(IALG_Handle handle,IALG_MemRec memTab[], IALG_Handle p, IALG_Params *algParams)
{
        ImageObj *img = (ImageObj *)handle;
        IMG_Params *params = (IMG_Params *)algParams;
        if (params == NULL) {
            params = &IMG_PARAMS; /* Use interface default parameters */
        }
        /* Initialize the logical DMA channel */
        img->dmaHandle_0 = NULL;
        img->dmaHandle_1 = NULL;
        img->dmaHandle_2 = NULL;
        img->grayScale = params->grayScale;
        return (IALG_EOK);
}
Int dmaGetChannels(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
        ImageObj *img = (ImageObj *)handle;
        /* algInit() is called before this fxn, so the `grayScale' field */
        /* in the instance object is initialized. */
        return (myDmaTabInit(&img, &dmaTab));
}
static Int myDmaTabInit(ImageObj *img, IDMA2_ChannelRec dmaTab[])
{
```

```
        dmaTab[0].handle = img->dmaHandle_0;
        /* If the image is grayscale, require just one logical DMA channel, */
        /* otherwise request three logical channels; one for each color plane */
        if (!img->grayScale) {
               dmaTab[1].handle = img->dmaHandle_1;
               dmaTab[2].handle = img->dmaHandle_2;
               /*
               * Transfers on each logical channel are independent.
               * i.e., each logical channel can be assigned to separate hardware queue.
               */
               dmaTab[0].queueId = 0;
               dmaTab[1].queueId = 1;
               dmaTab[2].queueId = 2;
               return (3);
        }
        return (1)
}
```

See Also

`dmaGetChannelCnt()`

1.3.6 dmaInit - grant the algorithm DMA resources

Synopsis

For IDMA2 interface

```
status = dmaInit(IALG_Handle handle, IDMA2_ChannelRec dmaTab[]);
```

For IDMA3 interface

```
status = dmaInit(IALG_Handle handle, IDMA3_ChannelRec dmaTab[]);
```

Arguments

For IDMA2 interface

```
IALG_Handle handle;        /* handle to algorithm instance */
IDMA2_ChannelRec dmaTab[]; /* input array of dma records */
```

For IDMA3 interface

```
IALG_Handle handle;        /* handle to algorithm instance */
IDMA3_ChannelRec dmaTab[]; /* input array of dma records */
```

Return Values

```
Int status;       /* Status indicating success or failure */
```

Description

dmaInit( ) performs all initialization of the algorithm instance pointed to by handle necessary for using the DMA resource. After a successful return from dmaInit( ), the algorithm instance is ready to use the DMA resource.

The first argument to dmaInit( ) is the algorithm's instance handle.

The second argument to dmaInit( ) is a table of dma records that describe the DMA resources. The handle field in the dmaTab[ ] array must be initialize by the client of the algorithm to contain a value which indicates a logical channel.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer for the algorithm's instance object.
- The handle field in the dmaTab[ ] array must be initialized.
- dmaInit( ) can only be called after a successful return from algInit( ).

Postconditions

The following conditions are true immediately after returning from this method.

- The algorithm object pointed to by handle has initialized its instance with the DMA resources passed in through dmaTab[ ].
- The algActivate() operation can be called.

Example

For IDMA2

```
 typedef struct ImageObj {
        IALG_Obj ialg; /* IALG object MUST be first field */
        IDMA2_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
        IDMA2_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
        IDMA2_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;
Int dmaInit(IALG_Handle handle, IDMA2_ChannelRec dmaTab[])
{
        ImageObj *img = (ImageObj *)handle;
        img->dmaHandle_0 = dmaTab[0].handle;
        /* algInit() is called before this fxn, so the `grayScale' field */
        /* in the instance object is initialized. */
        if (!img->grayScale) {
               img->dmaHandle_1 = dmaTab[1].handle;
```

```
                img->dmaHandle_2 = dmaTab[2].handle;
        }
        /* Additional algorithm initialization related to the DMA resorce */
       ....
       ....
        return (IALG_EOK);
}
```

For IDMA3

```
 typedef struct ImageObj {
        IALG_Obj ialg; /* IALG object MUST be first field */
        IDMA3_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
        IDMA3_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
        IDMA3_Handle dmaHandle_2 /* Handle for logical DMA channel 2 */
        Bool grayScale; /* TRUE = grayscale image, FALSE = RGB image */
} ImageObj;
Int dmaInit(IALG_Handle handle, IDMA3_ChannelRec dmaTab[])
{
        ImageObj *img = (ImageObj *)handle;
        img->dmaHandle_0 = dmaTab[0].handle;
        /* algInit() is called before this fxn, so the `grayScale' field */
        /* in the instance object is initialized. */
        if (!img->grayScale) {
               img->dmaHandle_1 = dmaTab[1].handle;
               img->dmaHandle_2 = dmaTab[2].handle;
        }
        /* Additional algorithm initialization related to the DMA resorce */
        ....
        ....
        return (IALG_EOK);
}
```

See Also

`dmaGetChannels(), dmaChangeChannels()`

ACPY2 Runtime APIs

This chapter describes the semantics of the ACPY2 APIs. These APIs can be called by an algorithm that has implemented the IDMA2 interface. A system using an algorithm that has implemented the IDMA2 interface must implement all these APIs.

Algorithms that have already been developed using the deprecated IDMA and ACPY APIs remain eXpressDSP-compliant; however, development of new algorithms should follow the new IDMA2/ACPY2 specification.

Topic **Page**

2.1 ACPY2 Functions

The following table provides a list of the ACPY2 functions and their descriptions.

Table 2-1. ACPY2 Functions

Function	Descriptions
ACPY2_complete()	Checks to see if the data transfers on a specific logical channel have completed
ACPY2_configure()	Configures a logical channel
ACPY2_exit()	Module finalization. Frees resources used by the ACPY module.
ACPY2_init()	Module initialization. Initializes the ACPY module.
ACPY2_setNumFrames()	Configures only the numFrames parameter of an IDMA channel
ACPY2_setSrcFrameIndex()	Configures only the source frame index parameter of an IDMA channel
ACPY2_setDstFrameIndex()	Configures only the dest frame index parameter of an IDMA channel
ACPY2_start()	Submits an asynchronous data transfer request
ACPY2_wait()	Waits for all data transfers to complete on a specific logical channel
ACPY2_getChanObjSize	Gets the size of the IDMA channel object
ACPY2_initChannel	Initializes the IDMA2 channel
ACPY2_startAligned()	Submits an asynchronous data transfer request

2.2 ACPY2_complete - non-blocking method to test DMA completion status

Synopsis

```
dmaDone = ACPY2_complete(IDMA2_Handle handle);
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
```

Return Value

```
Int  dmaDone;         /* dma completion flag */
```

Description

ACPY2_complete( ) checks to see if all the data transfers issued on the logical channel pointed to by handle have completed.

The only argument to ACPY2_complete( ) specifies the logical channel used for the data transfer requested with ACPY2_start( ) or ACPY2_startAligned( ).

The framework implementation of ACPY2_complete( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid handle to a granted logical DMA channel.

Postconditions

The following conditions are true immediately after returning from this method.

- If dmaDone = 0, the data transfer on the logical channel pointed to by handle are still in progress.
- If dmaDone != 0, the data transfer on the logical channel pointed to by handle have completed.

Examples

- Check to see if the data transfers the logical channel pointed to be handle have completed.

```
IDMA2_Handle  dmaHandle;
```

```
if (ACPY2_complete(dmaHandle) {
   startProcesingData();

}
else {

     `do some other work'

}
/* No more processing to do - wait for data transfers to complete */
ACPY2_wait(dmaHandle);
startProcesingData();
```

See Also

`ACPY2_wait(), ACPY2_StartAligned`

2.3 ACPY2_configure - Configure a logical DMA channel

Synopsis

```
Void ACPY2_configure(IDMA2_Handle handle, IDMA2_Params *params)
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
IDMA2_Params params;  /* Channel parameters */
```

Return Value

```
Void;
```

Description

ACPY2_configure( ) will set up the logical channel pointed to by handle with the values pointed to by params. An algorithm might call this API to prepare for repetitive DMA data transfers with the same configuration. The repetitive data transfers can then be executed faster.

The first argument to ACPY2_configure( ) specifies the logical channel subject to configuration.

The second argument to ACPY2_configure( ) points to the specific configuration parameters for the logical channel.

The ACPY2_configure implementation may choose to abort if a particular combination of transfer settings specified in the params argument is not supported in hardware or software. For example, C6x1x EDMA based implementation of ACPY2 may issue a SYS_abort if an algorithm attempts to set values of the frame or element indexes differently for source and destination, or if it requests element indexing for 2D transfers.

The framework implementation of ACPY2_configure( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- When element indexing is used, the value in srcElementIndex or dstElementIndex must be a multiple of the element size.
- A value of 0 in srcElementIndex or dstElementIndex disables element indexing.
- The transfer settings specified in the params argument must be supported in hardware or software by the client's implementation of ACPY2.
- If srcElementIndex is not equal to dstElementIndex then this feature must be supported by the target hardware of the ACPY2 implementation.
- If srcFrameIndex is not equal to dstFrameIndex then this feature must be supported by the target hardware of the ACPY2 implementation.
- params must be non-NULL

Postconditions

The following conditions are true immediately after returning from this method.

- The logical channel pointed to by handle is configured according to params.
- Frame index settings are ignored in 1D-to-1D transfers.
- ACPY2_start() or ACPY2_startAligned( ) can be called.

Examples

1. Configure the logical channel pointed to by handle for a 1D-to-1D transfer. We assume that the src and dst buffers are aligned on a 32-bit boundary. Note that the numFrames and frame index values will be ignored when xType=IDMA2_1D1D.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_1D1D;
params.elemSize = IDMA2_ELEM32;
params.numFrames = 0;    /* Not used in 1D1D transfer */
params.srcElemIndex = params.dstElemIndex = 0;
....
ACPY2_configure(dmaHandle,&params);
```

2. Configure the logical channel pointed to by handle for a 1D-to-2D transfer. We do not know if the src and dst for the transfer will be aligned, so we set the element size to 8 bits and do byte transfer. Let's say we want to transfer 8 frames and the "jump" between the end of a frame to the beginning of the next frame is 100 elements.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_1D2D;
params.elemSize = IDMA2_ELEM8;
params.numFrames = 8;
params.srcElemIndex = params.dstElemIndex = 0;
params.dstFrameIndex = 100;       /* In 8-bit bytes */
....
ACPY2_configure(dmaHandle,&params);
```

See Also

`ACPY2_start()`

2.4 ACPY2_configure - Configure a logical DMA channel

Synopsis

```
Void ACPY2_configure(IDMA2_Handle handle, IDMA2_Params *params)
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
IDMA2_Params params;  /* Channel parameters */
```

Return Value

```
Void;
```

Description

ACPY2_configure( ) will set up the logical channel pointed to by handle with the values pointed to by params. An algorithm might call this API to prepare for repetitive DMA data transfers with the same configuration. The repetitive data transfers can then be executed faster.

The first argument to ACPY2_configure( ) specifies the logical channel subject to configuration.

The second argument to ACPY2_configure( ) points to the specific configuration parameters for the logical channel.

The ACPY2_configure implementation may choose to abort if a particular combination of transfer settings specified in the params argument is not supported in hardware or software. For example, C6x1x EDMA based implementation of ACPY2 may issue a SYS_abort if an algorithm attempts to set values of the frame or element indexes differently for source and destination, or if it requests element indexing for 2D transfers.

The framework implementation of ACPY2_configure( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- When element indexing is used, the value in srcElementIndex or dstElementIndex must be a multiple of the element size.
- A value of 0 in srcElementIndex or dstElementIndex disables element indexing.
- The transfer settings specified in the params argument must be supported in hardware or software by the client's implementation of ACPY2.
- If srcElementIndex is not equal to dstElementIndex then this feature must be supported by the target hardware of the ACPY2 implementation.
- If srcFrameIndex is not equal to dstFrameIndex then this feature must be supported by the target hardware of the ACPY2 implementation.
- params must be non-NULL

Postconditions

The following conditions are true immediately after returning from this method.

- The logical channel pointed to by handle is configured according to params.
- Frame index settings are ignored in 1D-to-1D transfers.
- ACPY2_start() or ACPY2_startAligned( ) can be called.

Examples

1. Configure the logical channel pointed to by handle for a 1D-to-1D transfer. We assume that the src and dst buffers are aligned on a 32-bit boundary. Note that the numFrames and frame index values will be ignored when xType=IDMA2_1D1D.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_1D1D;
params.elemSize = IDMA2_ELEM32;
params.numFrames = 0;    /* Not used in 1D1D transfer */
params.srcElemIndex = params.dstElemIndex = 0;
....
ACPY2_configure(dmaHandle,&params);
```

2. Configure the logical channel pointed to by handle for a 1D-to-2D transfer. We do not know if the src and dst for the transfer will be aligned, so we set the element size to 8 bits and do byte transfer. Let's say we want to transfer 8 frames and the "jump" between the end of a frame to the beginning of the next frame is 100 elements.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_1D2D;
params.elemSize = IDMA2_ELEM8;
params.numFrames = 8;
params.srcElemIndex = params.dstElemIndex = 0;
params.dstFrameIndex = 100;       /* In 8-bit bytes */
....
ACPY2_configure(dmaHandle,&params);
```

See Also

`ACPY2_start()`

2.5 ACPY2_setNumFrames - configure transfer settings for number of frames

Synopsis

```
Void ACPY2_setNumFrames(IDMA2_Handle handle, Uns numFrames)
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
Uns  numFrames;       /* Channel transfer parameter */
```

Return Values

```
Void;
```

Description

ACPY2_setNumFrames( ) will quickly configure the number of frames per block setting of the logical channel pointed to by handle with the value passed in numFrames. An algorithm might call this API to prepare for DMA transfers using a different number of frames setting, while retaining all other transfer settings of the current channel's configuration.

The first argument to ACPY2_configure( ) specifies the logical channel subject to configuration.

The second argument to ACPY2_configure( ) holds the number of frames per block configuration parameter for the logical channel.

The framework implementation of ACPY2_setNumFrames() must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- the channel must be configured with at least one ACPY2_configure call.
- numFrames must be positive for 2D transfers.

Postconditions

The following conditions are true immediately after returning from this method.

- The logical channels number of frames setting is set to numFrames. Old values of all other configuration settings are retained.

Examples

First configure the logical channel pointed to by handle for a 1D-to-2D transfer. We do not know if the src and dst for the transfer will be aligned, so we must set the element size to 8 bits and do byte transfer. Initial configure call sets the number of frames to 8 and the "jump" between the end of a frame to the beginning of the next frame is 100 elements. The ACPY2_setNumFrames call then changes the number of frames to 10 for subsequent transfers, while keeping all other transfer settings.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_1D2D;
params.elemSize = IDMA2_ELEM8;
params.numFrames = 8;
params.dstFrameIndex = 100;

ACPY2_configure(dmaHandle,&params);
…0
ACPY2_setNumFrames(dmaHandle, 10);
```

See Also

```
ACPY2_configure()
```

2.6 ACPY2_setSrcFrameIndex - configure source frame index settings

Synopsis

```
Void ACPY2_setSrcFrameIndex(IDMA2_Handle handle, Int frameIndex)
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
Uns  frameIndex;      /* Channel transfer parameter */
```

Return Value

```
Void;
```

Description

ACPY2_setSrcFrameIndex( ) will quickly configure the source frame index setting of the logical channel pointed to by handle with the value passed in frameIndex. An algorithm might call this API to prepare for DMA transfers using a different setting for the source frame index, without changing any other transfer settings of the current configuration. For targets where DMA hardware does not support separate source and destination frame indexing the function configures the single frame index shared between the source and the destination.

The first argument to ACPY2_setSrcFrameIndex( ) specifies the logical channel subject to configuration.

The second argument to ACPY2_setSrcFrameIndex( ) holds the source frame index value that is used to set the logical channel's configuration.

The framework implementation of ACPY2_setSrcFrameIndex( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- the channel must be configured with at least one ACPY2_configure call.

Postconditions

- the logical channel's *source frame index* setting is changed to frameIndex. Old values of all other configuration settings are retained.
- if the target hardware supports a single *frame index* that is shared by both the source and destination configurations, the shared frame index is configured using the frameIndex arg.

Examples

Configure a logical DMA channel for 2D-to-2D transfers, where each block consists of 18 frames of 16-bits elements that are contiguous, with a gap of 20 bytes between two frames. Then change the source frame index to 10, forcing the subsequent transfers to use 10 byte wide spacing between frames.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_2D2D;
params.elemSize = IDMA2_ELEM16;
params.numFrames = 18;
params.srcFrameIndex = params.dstFrameIndex = 20;
params.srcElementIndex = params.dstElementIndex = 0;
ACPY2_configure(dmaHandle, &params);
. . .
ACPY2_setSrcFrameIndex (dmaHandle, 10);
```

See Also

```
ACPY2_configure()
```

2.7 ACPY2_setDstFrameIndex - configure destination frame index settings

Synposis

```
Void ACPY2_setDstFrameIndex(IDMA2_Handle handle, Uns frameIndex)
```

Arguments

```
IDMA2_Handle handle;  /* handle to a logical DMA channel */
Uns  frameIndex;      /* Channel transfer parameter */
```

Return Value

```
Void;
```

Description

ACPY2_setDstFrameIndex( ) will quickly configure the destination frame index setting of the logical channel pointed to by handle with the value passed in frameIndex. An algorithm might call this API to prepare for DMA transfers using a different setting for the destination frame index, without changing any other transfer settings of the current configuration. For targets where DMA hardware does not support separate source and destination frame indexing the function configures the single frame index shared between the source and the destination.

The first argument to ACPY2_setDstFrameIndex( ) specifies the logical channel subject to configuration.

The second argument to ACPY2_setDstFrameIndex( ) holds the destination frame index value that is used to set the logical channel's configuration.

The framework implementation of ACPY2_setDstFrameIndex( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- the channel must be configured with at least one ACPY2_configure call.

Postconditions

The following conditions are true immediately after returning from this method.

- the logical channel's *destination frame index* setting is changed to frameIndex. Old values of all other configuration settings are retained.
- if the target hardware supports a single *frame index* that is shared by both the source and destination configurations, the shared frame index is configured using the frameIndex arg.

Examples

Configure a logical DMA channel for 2D-to-2D transfers, where each block consists of 18 frames of 16-bits elements that are contiguous, with a gap of 20 bytes between two frames. Then change the destination frame index to 10, forcing the transfer to use 10 bytes wide spacing in destination frames.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_2D2D;
params.elemSize = IDMA2_ELEM16;
params.numFrames = 18;
params.srcFrameIndex = params.dstFrameIndex = 20;
params.srcElementIndex = params.dstElementIndex = 0;
ACPY2_configure(dmaHandle, &params);
 . . .
ACPY2_setDstFrameIndex (dmaHandle, 10);
```

See Also

```
ACPY2_configure()
```

2.8 ACPY2_start - submit a request for data transfer on a logical DMA channel

Synopsis

```
Void ACPY2_start(IDMA2_Handle handle, IDMA2_AdrPtr src, IDMA2_AdrPtr dst, Int cnt)
```

Arguments

```
IDMA2_Handle handle;/* handle to DMA resource*/
IDMA2_AdrPtr src;   /* Source address for data transfer */
IDMA2_AdrPtr dst;   /* Destination addr for data transfer */
Uns cnt;            /* Number of elements in a frame */
```

Descriptions

ACPY2_start( ) issues a request for a data transfer. The implementation of ACPY2_start( ) will copy these values to the appropriate DMA registers and start the data transfer, or put the request on a queue and program the DMA registers when the DMA is available.

The first argument to ACPY2_start( ) specifies the logical channel used for the data transfer as granted in dmaInit( ) or as changed with dmaChangeChannels( ). Repetitive requests for data transfers will take place in FIFO order.

The second argument to ACPY2_start( ) specifies the start address for the data transfer.

The third argument to ACPY2_start( ) specifies the destination address for the data transfer.

The fourth argument to ACPY2_start( ) indicates the number of elements that will be transferred from src to dst. In the case of a 2D transfer, cnt indicates the number of elements in each frame. The numFrames field in the params structure indicates how many frames of cnt elements that will be transferred. The total number of elements that will be transferred in the 2D case is then (cnt)x(numFrames).

The framework implementation of ACPY2_start( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- the logical channel must already be configured through ACPY2_configure( ).

Postconditions

The following conditions are true immediately after returning from this method.

- the channel configuration settings remains unchanged.
- the data transfer is in progress or in a queue waiting to be started.
- data transfer will not begin until after completion of **all** transfer requests that have been previously submitted to the logical channel indicated by the handle or to any other logical channel that shares the same queueId assigned by the same algorithm.

Examples

1. Start a DMA transfer from src to dst of 100 elements on a pre-configured logical channel.

   ```
   IDMA2_Handle  dmaHandle;
   IDMA2_AdrPtr  src, dst;
   ACPY2_start(dmaHandle, src, dst, 100);
   ```

2. Start a DMA transfer on a logical channel that currently has a different channel configuration. The transfer is 2D-to-2D, 16-bit elements, 32 contiguous elements in a frame, 18 frames and 20 elements between end of a frame to the start of the next frame.

   ```
   IDMA2_Handle  dmaHandle;

   IDMA2_Params  params;
   IDMA2_Handle  dmaHandle;
   params.xType = IDMA2_2D2D;
   params.elemSize = IDMA2_ELEM16;
   params.numFrames = 18;
   params.srcElemIndex = params.dstElemIndex = 0;
   /* 2 bytes per 16-bit elemen t */
   params.srcFrameIndex = params.dstFrameIndex = 20 * 2;
   ```

```
ACPY2_configure(dmaHandle, &params);
ACPY2_start(dmaHandle, src, dst, 32);
```

3. Start three DMA transfers on the same logical channel. The channel is pre-configured, and, all three transfers use the same configuration. The transfer is 2D-to-2D, 8-bit contiguous elements, 16 elements in a frame, 8 frames and 64 elements between end of a frame to the start of the next frame.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_2D2D;
params.elemSize = IDMA2_ELEM8;
params.numFrames = 8;
params.srcElemIndex = params.dstElemIndex = 0;
params.srcFrameIndex = params.dstFrameIndex = 64
ACPY2_configure(dmaHandle, &params);
ACPY2_start(dmaHandle, src1, dst1, 16);
ACPY2_start(dmaHandle, src2, dst2, 16);
ACPY2_start(dmaHandle, src3, dst3, 16);
```

See Also

`ACPY2_configure(), ACPY2_complete(), ACPY2_wait(), ACPY2_startAligned()`

2.9 ACPY2_wait - wait all transfers started on this logical DMA channel to finish

Synposis

```
Void ACPY2_wait(IDMA2_Handle handle);
```

Arguments

```
IDMA2_Handle handle;  /* handle to DMA resource*/
```

Return Value

`Void`

Description

ACPY2_wait( ) waits for all data transfer issues on the logical channel pointed to by handle to complete. After returning from ACPY2_wait(), all data transfer is guaranteed to be complete.

The only argument to ACPY2_wait( ) specifies the logical channel used for the data transfer requested with ACPY2_start( ) or ACPY2_startAligned( ).The framework implementation of ACPY2_wait( ) must be re-entrant.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid handle to a granted logical DMA channel.
- ACPY2_start or ACPY2_startAligned( ) must have been previously called on the handle.

Postconditions

The following conditions are true immediately after returning from this method.

- All data transfer on the logical channel pointed to by handle have completed.

Examples

Wait until all DMA data transfers are complete on the logical channel pointed to the handle.

```
IDMA2_Handle  dmaHandle;
ACPY2_wait(dmaHandle);
```

See Also

`ACPY2_start(),  ACPY2_complete(), ACPY2_startAligned()`

2.10 ACPY2_getChanObjSize - Return size of the handle for a logical channel

Synopsis

```
chanObjSize = ACPY2_getChanObjSize();
```

Arguments

```
none
```

Return Value

```
Uns  chanObjSize;     /* dma completion flag */
```

Descriptions

ACPY2_getChanObjSize( ) returns the size of the structure used by the underlying ACPY2 library implementation to represent a logical DMA handle where run-time channel state information can be stored.

This function can only be called by the client's framework and must not be called directly by the algorithm. The client uses the size information obtained by calling ACPY2_getChanObjSize( ) to allocate the memory needed for the DMA channel handle. The client subsequently must call ACPY2_initChannel( ) with the handle to initialize the logical channel's initial state. After its initialization, the handle can be granted to a requesting algorithm.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- none

Postconditions

- chanObjSize returned by the function is always > 0

Examples

Create a logical DMA handle. First inquire size of space needed to store handle, then allocate and initialize the channel associated with the handle.

```
    DMA2_ChannelRec dmaTab[_DMAN_MAXDMARECS];

    Int chanObjSize = ACPY2_getChanObjSize();

. . . .

   /* Allocate memory for channel handle */

   dmaTab[i].handle = (IDMA2_Handle)malloc(chanObjSize);

    /* Initialize channel state. Creation of the handle is complete */

    ACPY2_initChannel(dmaTab[i].handle, dmaTab[i].queueId);


    ACPY2_initChannel()
```

See Also

```
ACPY2_initChannel()
```

2.11 ACPY2_initChannel - Return size of the handle for a logical channel

Synposis

```
Void ACPY2_initChannel(IDMA2_Handle handle, Int queueId);
```

Arguments

```
IDMA2_Handle handle; /* handle to a logical DMA channel */
Int queueId;         /* logical queue identifier */
```

Return Values

```
Void;
```

Description

The ACPY2_initChannel( ) can only be called by the client's framework during the creation of a handle representing a logical DMA channel. It must not be called by the algorithm.

The queueId parameter is used to group several logical DMA channel together to share a single logical queue. The implementation ensures that the logical DMA channels that are initialized with the same value for queueId perform the DMA transfers in a strict first-in first-out (FIFO) fashion corresponding to the order which the DMA transfer requests are submitted to any of the channels.

The client may provide the same queueId information it receives from the algorithm or may apply a framework defined mapping of the algorithm assigned logical queue ids to framework queue ids. This mapping must preserve the algorithm's logical channel grouping requirements.

The client uses the size information obtained by calling ACPY2_getChanObjSize( ) to allocate the memory needed for the DMA channel handle. After initialization of the logical DMA channel by calling the ACPY2_initChannel( ) the handle can be granted to a requesting algorithm.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be non NULL and must point to private memory large enough to hold channel state (as obtained by ACPY2_getChanObjSize call).

Postconditions

The following conditions are true immediately after returning from this method.

- All channels initialized by the same queueId value belong to the same logical queue that strictly orders all DMA transfer requests submitted to any of the channels
- handle can be used to configure the logical DMA channel settings.

Examples

Create a logical DMA handle. First inquire size of space needed to store handle, then allocate and initialize the channel associated with the handle.

```
IDMA2_ChannelRec dmaTab[_DMAN_MAXDMARECS];
Int chanObjSize = ACPY2_getChanObjSize();
...
 * Allocate memory for channel handle */
dmaTab[i].handle = (IDMA2_Handle)malloc(chanObjSize);
  /* Initialize channel state. Creation of the handle is complete */
ACPY2_initChannel(dmaTab[i].handle, dmaTab[i].queueId);
ACPY2_getChanObjSize()
```

See Also

```
ACPY2_getChanObjSize()
```

2.12 ACPY2_startAligned - submit a request for data transfer on a logical DMA channel

Synopsis

```
Void ACPY2_startAligned(IDMA2_Handle handle, IDMA2_AdrPtr src, IDMA2_AdrPtr dst, Uns cnt)
```

Arguments

```
IDMA2_Handle handle;/* handle to DMA resource*/
IDMA2_AdrPtr src;   /* Source address for data transfer */
IDMA2_AdrPtr dst;   /* Destination addr for data transfer */
Uns cnt;            /* Number of elements in a frame */
```

Description

ACPY2_startAligned( ) issues a request for a data transfer. The implementation of ACPY2_startAligned( ) will copy these values to the appropriate DMA registers and start the data transfer, or put the request on a queue and program the DMA registers when the DMA is available.

The first argument to ACPY2_startAligned( ) specifies the logical channel used for the data transfer as granted in dmaInit( ) or as changed with dmaChangeChannels( ). Repetitive requests for data transfers will take place in FIFO order.

The second argument to ACPY2_startAligned( ) specifies the source address for the data transfer. The caller must ensure that the source address is aligned with respect to the element size.

The third argument to ACPY2_startAligned( ) specifies the destination address for the data transfer. The caller must ensure that the destination address is aligned with respect to the element size.

The fourth argument to ACPY2_startAligned( ) indicates the number of elements that will be transferred from src to dst. In the case of a 2D transfer, cnt indicates the number of elements in each frame. The numFrames field in the params structure indicates the number of frames of cnt elements that will be transferred. The total number of elements that will be transferred in the 2D case is then (cnt)x(numFrames).

The framework implementation of ACPY2_startAligned( ) must be re-entrant.

Note: The only operational difference between ACPY2_startAligned( ) and ACPY2_start( ) is the additional requirement by ACPY2_startAligned( ) for its source and destination addresses to be properly aligned with respect to the configured element size. For 32-bit transfers, these addresses must be at least 32-bit aligned. For 16-bit transfers, 16-bit alignment is required. When called with properly aligned addresses, both functions implement an identical behavior. However, in architectures (such as C6000) which permit DMA transfers using 8-bit or 16-bit alignment of src or dst addresses irrespective of the actual transfer element size, the ACPY2_startAligned( ) function can be optimized to operate more efficiently.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- handle must be a valid pointer to a granted logical DMA channel.
- src and dst addresses are aligned with respect to the transfer element size
- the logical channel must already be configured through ACPY2_configure( ).

Postconditions

The following conditions are true immediately after returning from this method.

- the channel configuration settings remains unchanged.
- the data transfer is in progress or in a queue waiting to be started.
- data transfer will not begin until after completion of all transfer requests that have been previously submitted to the logical channel indicated by the handle or to any other logical channel that shares the same queueId assigned by the same algorithm.

Examples

Start a DMA transfer from src to dst of 100 elements on a preconfigured logical channel.

```
IDMA2_Handle  dmaHandle;
IDMA2_AdrPtr  src, dst;
ACPY2_startAligned(dmaHandle, src, dst, 100);
```

Start a DMA transfer on a logical channel that currently has a different channel configuration. The transfer is 2D-to-2D, 16-bits elements, 32 contiguous elements in a frame, 18 frames and 20 elements between end of a frame to the start of the next frame. The src and dst addresses are aligned at 16-bit boundaries.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_2D2D;
params.elemSize = IDMA2_ELEM16;
params.numFrames = 18;
params.srcElemIndex = params.dstElemIndex = 0;
params.srcFrameIndex = params.dstFrameIndex = 20 * 2;

ACPY2_configure(dmaHandle, &params);
ACPY2_startAligned(dmaHandle, src, dst, 32);
```

Start three DMA transfers on the same logical channel. The channel is pre-configured, and, all three transfers use the same configuration. The transfer is 2D-to-2D, 8-bit contiguous elements, 16 elements in a frame, 8 frames and 64 elements between end of a frame to the start of the next frame. Since transfer is in 8-bit element mode, there is no additional alignment requirement on source and destination addresses.

```
IDMA2_Params  params;
IDMA2_Handle  dmaHandle;
params.xType = IDMA2_2D2D;
params.elemSize = IDMA2_ELEM8;
params.numFrames = 8;
params.srcElemIndex = params.dstElemIndex = 0;
params.srcFrameIndex = params.dstFrameIndex = 64

ACPY2_configure(dmaHandle, &params);
ACPY2_startAligned(dmaHandle, src1, dst1, 16);
ACPY2_startAligned(dmaHandle, src2, dst2, 16);
ACPY2_startAligned(dmaHandle, src3, dst3, 16);
```

See Also

`ACPY2_getChanObjSize()`

Supplementary APIs

This chapter describes supplementary module APIs that are available to the clients of XDAIS algorithms but are *not* part of the core run-time support.

3.1 Introduction

These modules are logically part of an XDAIS framework and are provided to simplify the use and management of eXpressDSP-compliant algorithms. These APIs define a simple XDAIS run-time support library that is provided in the *TMS320 DSP Algorithm Standard Developer's Guide* (SPRU424).

- ALG - module for the creation of algorithm instance objects

These APIs and any run-time support library provided by the TMS320 DSP Algorithm Standard Developer's Guide are entirely optional. They are not required in any application that uses XDAIS algorithm components. They are provided to simplify the use of XDAIS components in applications.

The relationship of these interfaces to the abstract interfaces defined in the previous chapter is illustrated by the figure below.

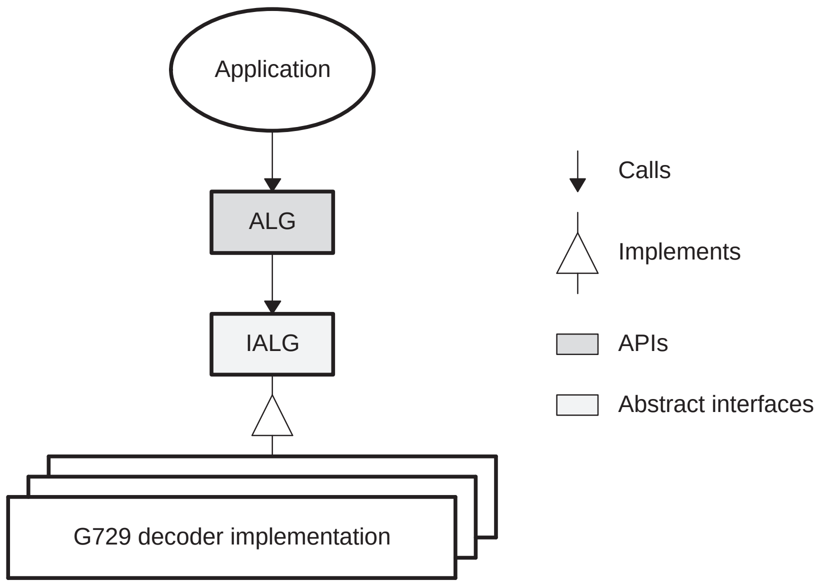


Figure 3-1. Abstract Interfaces and Module Interfaces

Every abstract interface corresponds to an API module that provides a conventional functional interface to any modules that implement the abstract interface. With the exception of the ALG module, these API modules contain little or no code; most operations are type-safe inline functions.

3.2 ALG - Algorithm Instance Object Manager

Synopsis

```
#include <alg.h>
```

Interface

```
/*--------------------------*/
/*    TYPES AND CONSTANTS    */
/*--------------------------*/
typedef IALG_Handle ALG_Handle;
/*--------------------------*/
/*        FUNCTIONS          */
/*--------------------------*/
ALG_activate();   /* initialize instance's scratch memory */
ALG_control();    /* send control command to algorithm */
ALG_create();     /* create an algorithm instance object */
ALG_deactivate(); /* save instance's persistent state */
ALG_delete();     /* delete algorithm instance's object */
ALG_exit();       /* ALG module finalization */
ALG_init();       /* ALG module initialization */
```

Description

The ALG module provides a generic (universal) interface used to create, delete, and invoke algorithms on data. The functions provided by this module use the IALG interface functions to dynamically create and delete algorithm objects. Any module that implements the IALG interface can be used by ALG.

The TMS320 DSP Developer's Kit includes several different implementations of the ALG module each implementing a different memory management policy. Each implementation optimally operates in a specified environment. For example, one implementation never frees memory; it should only be used in applications that never need to delete algorithm objects.

3.3 ALG_activate - initialize scratch memory buffers prior to processing

Synopsis

```
Void ALG_activate(ALG_Handle handle);
```

Arguments

```
ALG_Handle    handle;   /* algorithm instance handle */
```

Return Value

```
Void
```

Description

ALG_activate( ) initializes any scratch buffers and shared persistent memory using the persistent memory that is part of the algorithm's instance object. In preemptive environments, ALG_activate( ) saves all shared data memory used by this instance to a shadow memory so that it can be restored by ALG_deactivate( ) when this instance is deactivated.

The first (and only) argument to ALG_activate( ) is an algorithm instance handle. This handle is used by the algorithm to identify the various buffers that must be initialized prior to any processing methods being called.

See Also

```
ALG_deactivate()
```

3.4 ALG_create - create an algorithm object

Synopsis

```
handle = ALG_create(IALG_Fxns *fxns, IALG_Params *params);
```

Arguments

```
IALG_Fxns   *fxns;    /* pointer to algorithm functions */IALG_Params
*params; /* pointer to algorithm parameters */
```

Return Value

```
ALG_Handle   handle;  /* non-NULL handle of new object */
```

Description

ALG_create( ) implements a memory allocation policy and uses this policy to create an instance of the algorithm specified by fxns. The params parameter is a pointer to an algorithm-specific set of instance parameters that are required by the algorithm to create an instance.

If the return value of ALG_create( ) is NULL then it failed; otherwise the handle is non-NULL.

Example

```
#include <alg.h>
#include <encode.h>

Void main()
{
      ENCODE_Params        params;
      ALG_Handle           encoder;

      params = ENCODE_PARAMS;        /* initialize to default values */
      params.frameLen = 64;          /* set frame length */

      /* create instance of encoder object */
      encoder = ALG_create(&ENCODE_TI_IALG, (IALG_Params *)&params);

      if (encoder != NULL) {
         /* use encoder to encode data */
            ...
      }

      /* delete encoder object */
      ALG_delete(encoder);
```

See Also

```
ALG_delete()
```

3.5 ALG_control - send control command to algorithm

Synopsis

```
ret = ALG_control(ALG_Handle handle, ALG_Cmd cmd, ALG_Status *status);
```

Arguments

```
ALG_Handle   handle;   /* algorithm instance handle */
ALG_Cmd      cmd;      /* algorithm specific command */
ALG_Status   *status;  /* algorithm specific in/out buffer */
```

Return Value

```
Int ret;             /* return status (IALG_EOK, 0) */
```

Description

ALG_control( ) sends an algorithm specific command, cmd, and a pointer to an input/output status buffer pointer to an algorithm's instance object.

The first argument to ALG_control( ) is an algorithm instance handle. The second two parameters are interpreted in an algorithm-specific manner by the implementation.

The return value of ALG_control( ) indicates whether the control operation completed successfully. A return value of IALG_EOK indicates that the operation completed successfully; all other return values indicate failure.

Example

```
Void main()
{
      ALG_Handle encoder;
      ENCODE_Status status;

      /* create instance of encoder object */
      encoder = …;

      /* tell coder to minimize MIPS */
      status.u.mips = ENCODE_LOW
      ALG_control(encoder, ENCODE_SETMIPS, (ALG_Status *)&status);
      …
}
```

See Also

`ALG_control(), ALG_create()`

3.6 ALG_deactivate - save all persistent data to non-scratch memory

Synopsis

```
Void ALG_deactivate(ALG_Handle handle);
```

Arguments

```
ALG_Handle handle;   /* algorithm instance handle */
```

Return Value

```
Void                          /* none */
```

Description

ALG_deactivate( ) saves any persistent information to non-scratch buffers using the persistent memory that is part of the algorithm's instance object. In preemptive environments, ALG_deactivate( ) also restores any data previously saved to shadow memory by ALG_activate( ).

The first (and only) argument to ALG_deactivate( ) is an algorithm instance handle. This handle is used by the algorithm to identify the various buffers that must be saved prior to the next cycle of ALG_activate( ) and data processing calls.

See Also

`ALG_activate()`

3.7 ALG_delete - delete an algorithm object

Synopsis

```
Void ALG_delete(ALG_Handle handle);
```

Arguments

```
ALG_Handle handle;      /* algorithm instance handle */
```

Return Value

```
Void                          /* none */
```

Description

ALG_delete( ) deletes the dynamically created object referenced by handle, where handle is the return value from a previous call to ALG_create( ). If handle is NULL, ALG_delete( ) simply returns.

See Also

`ALG_create()`

3.8 ALG_init - module initialization

Synopsis

```
Void ALG_init(VOID);
```

Arguments

```
Void                  /* none */
```

Return Value

```
Void                  /* none */
```

Description

ALG_init( ) is called during system startup to perform any run-time initialization necessary for the algorithm module as a whole.

See Also

`ALG_create()`

3.9 ALG_exit - module clean- up

Synopsis

```
Void ALG_init(VOID);
```

Arguments

```
Void                  /* none */
```

Return Value

```
Void                  /* none */
```

Description

ALG_init( ) is called during system startup to perform any run-time initialization necessary for the algorithm module as a whole.

See Also

`ALG_delete()`

Example Algorithm Implementation

This appendix contains the complete source code to two eXpressDSP-compliant algorithm modules; a finite impulse response filter module (FIR) and a filter group module (FIG).

A.1 FIR Filter Example

Although a digital filter is much too simple an algorithm to encapsulate as an XDAIS component, it illustrates (and hopefully motivates) the concepts presented in the XDAIS specification. The FIR filter example consists of the following files:

1. *fir.c, fir.h* - FIR utility API module source and interface header
2. *ifir.c, ifir.h* - abstract FIR interface definition header and parameter defaults
3. *fir_ti.c, fir_ti.h* - vendor specific implementation and header
4. *fir_ti_ext.c* - vendor specific extensions to FIR
5. *firtest.c, firtest1.c* - simple programs using ALG to execute a FIR filter.

For simplicity, all of the IALG interface functions are implemented in a single file and the algorithm is written in C. Figure A-1 illustrates the relationship between the files used in this section.

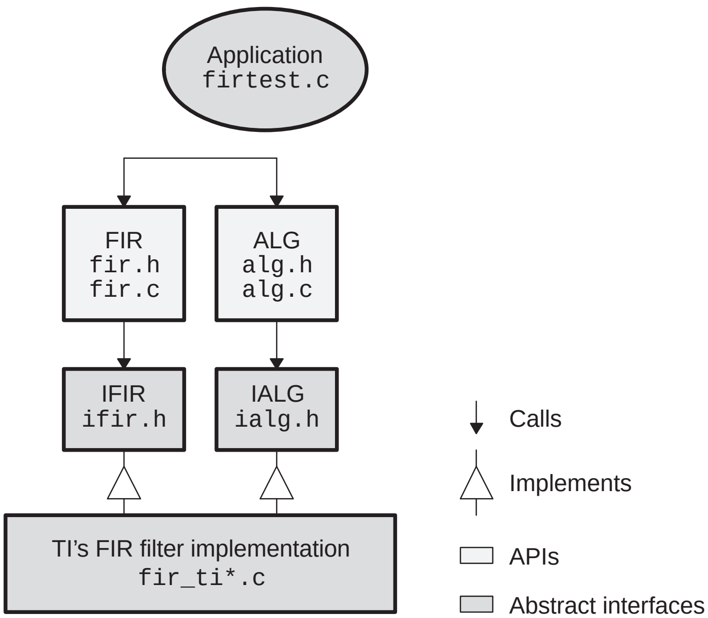


Figure A-1. FIR Filter Example Implementation

The filter group module, FIG, is an example that illustrates how multiple instances of an algorithm can be grouped together to share common coefficients.

The filter group example consists of the following files.

1. *fig.c, fig.h* - FIG utility API module source and interface header
2. *ifig.h* - abstract FIG interface definition header
3. *fig_ti.c, fig_ti.h* - vendor specific implementation and header
4. *figtest.c* - a simple program using ALG to execute a filter group.

In addition to providing the appropriate run-time interfaces, every eXpressDSP-compliant algorithm must also be accompanied by a characterization of its performance. The required metrics are described in the XDAIS specification and summarized in Appendix A. The spreadsheet below captures the relevant information for the FIR example.

Instance Parameters	
filterlen	16
framelen	180

Other Parameters	
word size (bytes)	2
sample rate (samp/sec)	8000

Execution Time	Period	Cycles/Period
worst case	22500 us	2880

Interrupt Latency	0 cycles

Stack Memory	Size	Align
worst case	40	0

Instance Memory	DARAM		SARAM		External	
	Size	Align	Size	Align	Size	Align
scratch	390	0	0	0	0	0
persistent	0	0	0	0	42	0

Module Memory	Code		Data		BSS	
	Size	Align	Size	Align	Size	Align
fir_ti.o54	734	0	0	0	34	0
fir_ti_ext.o54	134	0	0	0	0	0

A.2 fir.h - FIR Module Interface

Text

```
/*
 *  ======== fir.h ========
 *  This header defines all types, constants, and functions used by
 *  applications that use the FIR algorithm.
 *
 *  Applications that use this interface enjoy type safety and
 *  the ability to incorporate multiple implementations of the FIR
 *  algorithm in a single application at the expense of some
 *  additional indirection.
 */

#ifndef FIR_
#define FIR_

#include <alg.h>
#include <ifir.h>
#include <ialg.h>


/*
 *  ======== FIR_Handle ========
 *  FIR algorithm instance handle
 */
typedef struct IFIR_Obj *FIR_Handle;


/*
 *  ======== FIR_Params ========
 *  FIR algorithm instance creation parameters
 */
typedef struct IFIR_Params FIR_Params;


/*
 *  ======== FIR_PARAMS ========
 *  Default instance parameters
 */
#define FIR_PARAMS IFIR_PARAMS


/*
 *  ======== FIR_apply ========
 *  Apply a FIR filter to the input array and place results in the
 *  output array.
 */
extern Void FIR_apply(FIR_Handle fir, Int in[], Int out[]);
/*
/*
 *  ======== FIR_create ========
 *  Create an instance of a FIR object.
 */
static inline FIR_Handle FIR_create(const IFIR_Fxns *fxns,
                                    const FIR_Params *prms)
{
    return ((FIR_Handle)ALG_create((IALG_Fxns *)fxns,
                                    NULL, (IALG_Params *)prms));
}


/*
 *  ======== FIR_delete ========
 *  Delete a FIR instance object
 */
```

```
static inline Void FIR_delete(FIR_Handle handle)
{
    ALG_delete((ALG_Handle)handle);
}


/*
 *  ======== FIR_exit ========
 *  Module finalization
 */
extern Void FIR_exit(Void);

/*
 *  ======== FIR_init ========
 *  Module initialization
 */
extern Void FIR_init(Void);

#endif  /* FIR_ */
```

A.3 ifir.h - Example Abstract FIR Filter Interface

Text

```
/*
 *  ======== ifir.h ========
 *  This header defines all types, constants, and functions shared by all
 *  implementations of the FIR algorithm.
 */
#ifndef IFIR_
#define IFIR_

#include <ialg.h>

/*
 *  ======== IFIR_Obj ========
 *  Every implementation of IFIR *must* declare this structure as
 *  the first member of the implementation's object.
 */
typedef struct IFIR_Obj {
    struct IFIR_Fxns *fxns;
} IFIR_Obj;

/*
 *  ======== IFIR_Handle ========
 *  This type is a pointer to an implementation's instance object.
 */
typedef struct IFIR_Obj *IFIR_Handle;

/*
 *  ======== IFIR_Params ========
 *  This structure defines the parameters necessary to create an
 *  instance of a FIR object.
 *
 *  Every implementation of IFIR *must* declare this structure as
 *  the first member of the implementation's parameter structure.
 */
typedef struct IFIR_Params {
    Int size;           /* sizeof the whole parameter struct */
    Int *coeffPtr;      /* pointer to coefficients */
    Int filterLen;      /* length of filter */
    Int frameLen;       /* length of input (output) buffer */
} IFIR_Params;

/*
 *  ======== IFIR_PARAMS ========
```

```
 *  Default instance creation parameters (defined in ifir.c)
 */
extern IFIR_Params IFIR_PARAMS;

/*
 *  ======== IFIR_Fxns ========
 *  All implementation's of FIR must declare and statically
 *  initialize a constant variable of this type.
 *
 *  By convention the name of the variable is FIR_<vendor>_IFIR, where
 *  <vendor> is the vendor name.
 */
typedef struct IFIR_Fxns {
    IALG_Fxns   ialg;
    Void        (*filter)(IFIR_Handle handle, Int in[], Int out[]);
} IFIR_Fxns;

#endif  /* IFIR_ */
```

A.4 fir.c - Common FIR Module Implementation

Text

```
/*
 *  ======== fir.c ========
 *  FIR Filter Module - implements all functions and defines all constant
 *  structures common to all FIR filter algorithm implementations.
 */
#include <std.h>
#include <alg.h>

#include <fir.h>

/*
 *  ======== FIR_apply ========
 *  Apply a FIR filter to the input array and place results in the
 *  output array.
 */
Void FIR_apply(FIR_Handle handle, Int in[], Int out[])
{
    /* activate instance object */
    ALG_activate((ALG_Handle)handle);

    handle->fxns->filter(handle, in, out);      /* filter data */

    /* deactivate instance object */
    ALG_deactivate((ALG_Handle)handle);
}

/*
 *  ======== FIR_exit ========
 *  Module finalization
 */
Void FIR_exit()
{
}

/*
 *  ======== FIR_init ========
 *  Module initialization
 */
Void FIR_init()
{
}
```

A.5 fir_ti.c - Vender-Specific FIR Module Implementation

Text

```
/*
 *  ======== fir_ti.c ========
 *  FIR Filter Module - TI implementation of a FIR filter algorithm
 *
 *  This file contains an implementation of the IALG interface
 *  required by XDAIS.
 */
#pragma CODE_SECTION(FIR_TI_activate, ".text:algActivate")
#pragma CODE_SECTION(FIR_TI_alloc, ".text:algAlloc()")
#pragma CODE_SECTION(FIR_TI_deactivate, ".text:algDeactivate")
#pragma CODE_SECTION(FIR_TI_free, ".text:algFree")
#pragma CODE_SECTION(FIR_TI_initObj, ".text:algInit")
#pragma CODE_SECTION(FIR_TI_moved, ".text:algMoved")

#include <std.h>

#include <ialg.h>
```

```
#include <ifir.h>
#include <fir_ti.h>
#include <fir_ti_priv.h>

#include <string.h>        /* memcpy() declaration */

#define HISTORY 1
#define WORKBUF 2
#define NUMBUFS 3

/*
 *  ======== dot ========
 */
static Int dot(Int *a, Int *b, Int n)
{
    Int sum = 0;
    Int I;

    for (I = 0; I < n; I++) {
        sum += *a++ * *b++;
    }
    return (sum);
}

/*
 *  ======== FIR_TI_activate ========
 *  Copy filter history from external slow memory into working buffer.
 */
Void FIR_TI_activate(IALG_Handle handle)
{
    FIR_TI_Obj *fir = (Void *)handle;

    /* copy saved history to working buffer */
    memcpy((Void *)fir->workBuf, (Void *)fir->history,
        fir->filterLenM1 * sizeof(Int));
}

/*
 *  ======== FIR_TI_alloc ========
 */
Int FIR_TI_alloc(const IALG_Params *algParams,
                 IALG_Fxns **pf, IALG_MemRec memTab[])
{
    const IFIR_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IFIR_PARAMS;  /* set default parameters */
    }

    /* Request memory for FIR object */
    memTab[0].size = sizeof(FIR_TI_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_EXTERNAL;
    memTab[0].attrs = IALG_PERSIST;

    /*
     *  Request memory filter's "inter-frame" state (i.e., the
     *  delay history)
     *
     *  Note we could have simply added the delay buffer size to the
     *  end of the FIR object by combining this request with the one
     *  above, thereby saving some code.  We separate it here for
     *  clarity.
     */
    memTab[HISTORY].size = (params->filterLen - 1) * sizeof(Int);
    memTab[HISTORY].alignment = 0;
```

```
    memTab[HISTORY].space = IALG_EXTERNAL;
    memTab[HISTORY].attrs = IALG_PERSIST;

    /*
     *  Request memory for shared working buffer
     */
    memTab[WORKBUF].size =
        (params->filterLen - 1 + params->frameLen) * sizeof(Int);
    memTab[WORKBUF].alignment = 0;
    memTab[WORKBUF].space = IALG_DARAM0;
    memTab[WORKBUF].attrs = IALG_SCRATCH;

    return (NUMBUFS);
}



/*
 *  ======== FIR_TI_deactivate ========
 *  Copy filter history from working buffer to external memory
 */
Void FIR_TI_deactivate(IALG_Handle handle)
{
    FIR_TI_Obj *fir = (Void *)handle;

    /* copy history to external history buffer */
    memcpy((Void *)fir->history, (Void *)fir->workBuf,
        fir->filterLenM1 * sizeof(Int));
}

/*
 *  ======== FIR_TI_filter ========
 */
Void FIR_TI_filter(IFIR_Handle handle, Int in[], Int out[])
{
    FIR_TI_Obj *fir = (Void *)handle;
    Int *src = fir->workBuf;
    Int *dst = out;
    Int I;

    /* copy input buffer into working buffer */
    memcpy((Void *)(fir->workBuf + fir->filterLenM1), (Void *)in,
        fir->frameLen * sizeof (Int));

    /* filter data */
    for (I = 0; I < fir->frameLen; I++) {
        *dst++ = dot(src++, fir->coeff, fir->filterLenM1 + 1);
    }

    /* shift filter history to start of work buffer for next frame */
    memcpy((Void *)fir->workBuf, (Void *)(fir->workBuf + fir->frameLen),
        fir->filterLenM1 * sizeof (Int));
}
```

A.6 fir_ti.h - Vender-Specific FIR Module Interface

Text

```
/*
 *  ======== fir_ti.h ========
 *  Vendor specific (TI) interface header for FIR algorithm.
 *
 *  Applications that use this interface enjoy type safety and
 *  minimal overhead at the expense of being tied to a
 *  particular FIR implementation.
 *
```

```
 *  This header only contains declarations that are specific
 *  to this implementation.  Thus, applications that do not
 *  want to be tied to a particular implementation should never
 *  include this header (i.e., it should never directly
 *  reference anything defined in this header.)
 */
#ifndef FIR_TI_
#define FIR_TI_

#include <ialg.h>
#include <ifir.h>

/*
 *  ======== FIR_TI_exit ========
 *  Required module finalization function
 */
extern Void FIR_TI_exit(Void);

/*
 *  ======== FIR_TI_init ========
 *  Required module initialization function
 */
extern Void FIR_TI_init(Void);

/*
 *  ======== FIR_TI_IALG ========
 *  TI's implementation of FIR's IALG interface
 */
extern IALG_Fxns FIR_TI_IALG;

/*
 *  ======== FIR_TI_IFIR ========
 *  TI's implementation of FIR's IFIR interface
 */
extern IFIR_Fxns FIR_TI_IFIR;

/*
 *  ======== Vendor specific methods  ========
 *  The remainder of this file illustrates how a vendor can
 *  extend an interface with custom operations.
 *
 *  The operations below simply provide a type safe interface
 *  for the creation, deletion, and application of TI's FIR filters.
 *  However, other implementation specific operations can also
 *  be added.
 */

/*
 *  ======== FIR_TI_Handle ========
 */
typedef struct FIR_TI_Obj *FIR_TI_Handle;

/*
 *  ======== FIR_TI_Params ========
 *  We don't add any new parameters to the standard ones defined
 *  by IFIR.
 */
typedef IFIR_Params FIR_TI_Params;

/*
 *  ======== FIR_TI_PARAMS ========
 *  Define our defult parameters.
 */
#define FIR_TI_PARAMS   IFIR_PARAMS

/*
```

```
 *  ======== FIR_TI_create ========
 *  Create a FIR_TI instance object.
 */
extern FIR_TI_Handle FIR_TI_create(const FIR_TI_Params *params);

/*
 *  ======== FIR_TI_delete ========
 *  Delete a FIR_TI instance object.
 */
extern Void FIR_TI_delete(FIR_TI_Handle handle);


/*
 *  ======== FIR_TI_nApply ========
 *  Apply specified FIR filter to n input frames and overwrite
 *  input with the result.
 */
extern Void FIR_TI_nApply(FIR_TI_Handle handle, Int inout[], Int n);

#endif  /* FIR_TI_ */
```

A.7 fir_ti_priv.h - Private Vender-Specific FIR Header

Text

```
/*
 *  ======== fir_ti_priv.h ========
 *  Internal vendor specific (TI) interface header for FIR
 *  algorithm. Only the implementation source files include
 *  this header; this header is not shipped as part of the
 *  algorithm.
 *
 *  This header contains declarations that are specific to
 *  this implementation and which do not need to be exposed
 *  in order for an application to use the FIR algorithm.
 */
#ifndef FIR_TI_PRIV_
#define FIR_TI_PRIV_

#include <ialg.h>
#include <ifir.h>

typedef struct FIR_TI_Obj {
    IALG_Obj    alg;            /* MUST be first field of XDAIS algs */
    Int         *workBuf;       /* on-chip scratch history */
    Int         *coeff;         /* on-chip persistant coeff */
    Int         *history;       /* off chip persistant history */
    Int         filterLenM1;    /* length of coefficient array - 1 */
    Int         frameLen;       /* length of input (output) buffer */
} FIR_TI_Obj;



extern Void FIR_TI_activate(IALG_Handle handle);

extern Void FIR_TI_deactivate(IALG_Handle handle);

extern Int FIR_TI_alloc(const IALG_Params *algParams, IALG_Fxns **pf,
                        IALG_MemRec memTab[]);

extern Int FIR_TI_free(IALG_Handle handle, IALG_MemRec memTab[]);

extern Int FIR_TI_initObj(IALG_Handle handle,
                          const IALG_MemRec memTab[], IALG_Handle parent,
                          const IALG_Params *algParams);
```

```
extern Void FIR_TI_moved(IALG_Handle handle,
                          const IALG_MemRec memTab[], IALG_Handle parent,
                          const IALG_Params *algParams);

extern Void FIR_TI_filter(IFIR_Handle handle, Int in[], Int out[]);

#endif  /* FIR_TI_PRIV_ */
```

A.8 fir_ti_ext.c - Vender-Specific FIR Extensions

Text

```
/*
 *  ======== fir_ti_ext.c ========
 */
#pragma CODE_SECTION(FIR_TI_create, ".text:create")
#pragma CODE_SECTION(FIR_TI_delete, ".text:delete")
#pragma CODE_SECTION(FIR_TI_init, ".text:init")
#pragma CODE_SECTION(FIR_TI_exit, ".text:exit")

#include <std.h>
#include <alg.h>
#include <ialg.h>
#include <fir.h>
#include <ifir.h>

#include <fir_ti.h>
#include <fir_ti_priv.h>

/*
 *  ======== FIR_TI_create ========
 */
FIR_TI_Handle FIR_TI_create(const FIR_Params *params)
{
    return ((Void *)ALG_create(&FIR_TI_IALG,NULL,(IALG_Params *)params));
}

/*
 *  ======== FIR_TI_delete ========
 */
Void FIR_TI_delete(FIR_TI_Handle handle)
{
    ALG_delete((ALG_Handle)handle);
}

/*
 *  ======== FIR_TI_exit ========
 */
Void FIR_TI_exit(Void)
{
    ALG_exit();
}


/*
 *  ======== FIR_TI_init ========
 */
Void FIR_TI_init(Void)
{
    ALG_init();
}


/*
 *  ======== FIR_TI_nApply ========
 */
Void FIR_TI_nApply(FIR_TI_Handle handle, Int input[], Int n)
{
    Int *in;
    Int i;

    ALG_activate((ALG_Handle)handle);

    for (in = input, i = 0; i < n; i++) {
        FIR_TI_filter((IFIR_Handle)handle, in, in);
```

```
        in += handle->frameLen;
    }

    ALG_deactivate((ALG_Handle)handle);
}
```

A.9 fir_ti_ifirvt.c - Vendor-Specific IFIR Function Table

Text

```
/*
 *  ======== fir_ti_ifirvt.c ========
 *  This file contains the function table definitions for all
 *  interfaces implemented by the FIR_TI module that derive
 *  from IALG
 *
 *  We place these tables in a separate file for two reasons:
 *     1. We want to allow one to one to replace these tables
 *        with different definitions.  For example, one may
 *        want to build a system where the FIR is activated
 *        once and never deactivated, moved, or freed.
 *
 *     2. Eventually there will be a separate "system build"
 *        tool that builds these tables automatically
 *        and if it determines that only one implementation
 *        of an API exists, "short circuits" the vtable by
 *        linking calls directly to the algorithm's functions.
 */
#include <std.h>

#include <ialg.h>
#include <ifir.h>

#include <fir_ti.h>
#include <fir_ti_priv.h>

#define IALGFXNS \
    &FIR_TI_IALG,       /* module ID */                         \
    FIR_TI_activate,    /* activate */                          \
    FIR_TI_alloc,       /* alloc */                             \
    NULL,               /* control (NULL => no control ops) */  \
    FIR_TI_deactivate,  /* deactivate */                        \
    FIR_TI_free,        /* free */                              \
    FIR_TI_initObj,     /* init */                              \
    FIR_TI_moved,       /* moved */                             \
    NULL                /* numAlloc() (NULL => IALG_MAXMEMRECS) */\

/*
 *  ======== FIR_TI_IFIR ========
 *  This structure defines TI's implementation of the IFIR interface
 *  for the FIR_TI module.
 */
IFIR_Fxns FIR_TI_IFIR = {       /* module_vendor_interface */
    IALGFXNS,
    FIR_TI_filter    /* filter */
};

/*
 *  ======== FIR_TI_IALG ========
 *  This structure defines TI's implementation of the IALG interface
 *  for the FIR_TI module.
 */
#ifdef _TI_

asm("_FIR_TI_IALG .set _FIR_TI_IFIR");
```

```
#else

/*
 *  We duplicate the structure here to allow this code to be compiled and
 *  run non-DSP platforms at the expense of unnecessary data space
 *  consumed by the definition below.
 */
IALG_Fxns FIR_TI_IALG = {       /* module_vendor_interface */
    IALGFXNS
};

#endif
```

A.10 firtest.c - Example Client of FIR Utility Library

Text

```
/*
 *  ======== firtest.c ========
 *  This example shows how to use the type safe FIR "utility"
 *  library directly by an application.
 */
#include <std.h>
#include <fir.h>
#include <log.h>
#include <fir_ti.h>
#include <stdio.h>
extern LOG_Obj trace;
Int coeff[] = {1, 2, 3, 4, 4, 3, 2, 1};
Int input[] = {1, 0, 0, 0, 0, 0, 0};
#define FRAMELEN    (sizeof (input) / sizeof (Int))
#define FILTERLEN   (sizeof (coeff) / sizeof (Int))
Int output[FRAMELEN];
static Void display(Int a[], Int n);
/*
 *  ======== main ========
 */
Int main(Int argc, String argv[])
{
   {
   FIR_Params firParams;
   FIR_Handle fir;

   FIR_init();

   firParams = FIR_PARAMS;
   firParams.filterLen = FILTERLEN;
   firParams.frameLen = FRAMELEN;
   firParams.coeffPtr = coeff;
   if ((fir = FIR_create(&FIR_TI_IFIR, &firParams)) != NULL) {
       FIR_apply(fir, input, output);      /* filter some data */
       display(output, FRAMELEN);          /* display the result */
       FIR_delete(fir);                    /* delete the filter */
   }
    FIR_exit();
   return (0);
}

/*
*  ======== display ========
*/
static Void display(Int a[], Int n)
{
 Int i;
  for (i = 0; i < n; i++) {
```

```
            LOG_printf(&trace, "%d ", a[i]);
        }
        LOG_printf(&trace, "\n");
    }
```

A.11 firtest1.c - Example Client of ALG and FIR

Text

```
/*
*  ======== firtest1.c ========
 *  This example shows how to create an algorithm instance object
   using the ALG interface.
 *
*  The ALG interface allows one to create code that can create
 *  an instance of *any* XDAIS algorithm at the cost of a loss of
 *  type safety.
 */
#include <std.h>
#include <fir.h>
#include <alg.h>
#include <log.h>
#include <ialg.h>
#include <fir_ti.h>
extern LOG_Obj trace;
Int coeff[] = {1, 2, 3, 4, 4, 3, 2, 1};
Int input[] = {1, 0, 0, 0, 0, 0, 0};
#define FRAMELEN    (sizeof (input) / sizeof (Int))
#define FILTERLEN   (sizeof (coeff) / sizeof (Int))
Int output[FRAMELEN];
static Void display(Int a[], Int n);
/*
*  ======== main ========
 */
Int main(Int argc, String argv[])
{
    FIR_Params firParams;
    ALG_Handle alg;
    ALG_init();
    FIR_init();

    /* create an instance of a FIR algorithm */
    firParams = FIR_PARAMS;
    firParams = FIR_PARAMS;
    firParams.filterLen = FILTERLEN;
    firParams.frameLen = FRAMELEN
    firParams.coeffPtr = coeff;
    alg = ALG_create((IALG_Fxns *)&FIR_TI_IFIR, NULL,
                     (IALG_Params *)&firParams);
   /* if the instance creation succeeded, create a trace descriptor */
    if (alg != NULL) {
      FIR_apply((FIR_Handle)alg, input, output);  /* filter data */
      display(output, FRAMELEN);                  /* display result */
      ALG_delete(alg);        /* delete alg instance */
    }
    FIR_exit();
    ALG_exit();
    return (0);
}

/*
 *  ======== display ========
 */
static Void display(Int a[], Int n)
{
  Int i;
   for (i = 0; i < n; i++) {
       LOG_printf(&trace, "%d ", a[i]);
    }
    LOG_printf(&trace, "\n");
}
```

A.12 fig.h - Filter Group Module Interface

Text

```
/*
*  ======== fig.h ========
*  Filter Group Module Header - This module implements a FIR
*  filter group object.  A filter group object simply
 *  maintains global state (common coefficients and working
*  buffer) multiple FIR objects.  Thus, this module does not
 *  have a "process" method, it only implements "activate"
* "deactivate", and "getStatus".
 */
#ifndef FIG_
#define FIG_
#include <ifig.h>
typedef struct IFIG_Obj *FIG_Handle;
/*
*  ======== FIG_Params ========
 *  Filter group instance creation parameters
 */
typedef struct IFIG_Params FIG_Params;
extern const FIG_Params FIG_PARAMS; /* default instance parameters */
/*
 *  ======== FIG_Status ========
 *  Status structure for getting FIG instance attributes
 */
typedef struct IFIG_Status FIG_Status;
/*
 *  ======== FIG_activate ========
 */
extern Void FIG_activate(FIG_Handle handle);
/*
 *  ======== FIG_create ========
*/
extern FIG_Handle FIG_create(IFIG_Fxns *fxns, IFIG_Params *prms);
/*
 *  ======== FIG_deactivate ========
 */
extern Void FIG_deactivate(FIG_Handle handle);
/*
 *  ======== FIG_delete ========
 */
extern Void FIG_delete(FIG_Handle fir);
/*
 *  ======== FIG_getStatus ========
 */
extern Void FIG_getStatus(FIG_Handle fig, FIG_Status *status);
#endif  /* FIG_ */
```

A.13 ifig.h - Example Abstract FIR Filter Group Interface

Text

```
/*
 *  ======== ifig.h ========
 *  Filter Group Module Header - This module implements a FIR filter
 *  group object.  A filter group object simply maintains global state
 *  (common coefficients and working buffer) multiple FIR objects.
 *  Thus, this module does not have a "process" method, it only
 *  implements "activate" and "deactivate".
 */
#ifndef IFIG_
#define IFIG_
#include <ialg.h>
/*
*  ======== IFIG_Params ========
 *  Filter group instance creation parameters
 */
typedef struct IFIG_Params {
   Int size;           /* sizeof this structure */
    Int *coeffPtr;      /* pointer to coefficient array */
    Int filterLen;      /* length of coefficient array (words) */
} IFIG_Params;
extern const IFIG_Params IFIG_PARAMS;   /* default instance parameters */
/*
 *  ======== IFIG_Obj ========
 */
typedef struct IFIG_Obj {
    struct IFIG_Fxns *fxns;
} IFIG_Obj;
/*
 *  ======== IFIG_Handle ========
 */
typedef struct IFIG_Obj *IFIG_Handle;
/*
 *  ======== IFIG_Status ========
 *  Status structure for getting FIG instance attributes
 */
typedef struct IFIG_Status {
    Int *coeffPtr;          /* pointer to coefficient array */
} IFIG_Status;
/*
 *  ======== IFIG_Fxns ========
 */
typedef struct IFIG_Fxns {
    IALG_Fxns ialg;
    Void (*getStatus)(IFIG_Handle handle, IFIG_Status *status);
} IFIG_Fxns;
#endif  /* IFIG_ */
```

A.14 fig.c - Common Filter Group Module Implementation

Text

```
/*
 *  ======== fig.c ========
 *  Filter Group - this module implements a filter group; a group of FIR
*  filters that share a common set of coefficients and a working buffer.
*/
#include <std.h>
#include <fig.h>
/*
 *  ======== FIG_exit ========
*/
Void FIG_exit(Void)
{
}
/*
 *  ======== FIG_init ========
 */
Void FIG_init(Void)
{
}
```

A.15 fig.c - Vendor-Specific Filter Group Module Implementation

Text

```
/*
 *  ======== fig_ti.c ========
 *  Filter Group - this module implements a filter group; a group of FIR
 *  filters that share a common set of coefficients and a working buffer.
 */
#pragma CODE_SECTION(FIG_TI_alloc, ".text:algAlloc()")
#pragma CODE_SECTION(FIG_TI_free, ".text:algFree")
#pragma CODE_SECTION(FIG_TI_initObj, ".text:algInit")
#pragma CODE_SECTION(FIG_TI_moved, ".text:algMoved")
#include <std.h>
#include <ialg.h>
#include <fig_ti.h>
#include <ifig.h>
#include <string.h>     /* memcpy() declaration */
#define COEFF   1
#define NUMBUFS 2
typedef struct FIG_TI_Obj {
    IALG_Obj    alg;            /* MUST be first field of XDAIS algs */
    Int         *coeff;         /* on-chip persistant coefficient array */
    Int         filterLen;      /* filter length (in words) */
} FIG_TI_Obj;
/*
 *  ======== FIG_TI_alloc ========
*/
Int FIG_TI_alloc(const IALG_Params *algParams, IALG_Fxns **parentFxns,
                IALG_MemRec memTab[])
{
   const IFIG_Params *params = (Void *)algParams;
 if (params == NULL) {
        params = &IFIG_PARAMS;  /* set default parameters */
   }
   /* Request memory for FIG object */
   memTab[0].size = sizeof (FIG_TI_Obj);
   memTab[0].alignment = 0;
   memTab[0].space = IALG_EXTERNAL;
   memTab[0].attrs = IALG_PERSIST;
    /*
    *  Request memory for filter coefficients
    *
```

```
     *  Note that this buffer is declared as persistent; i.e., it is the
     *  responsibility of the client to insure that its contents are
     *  preserved whenever this object is active.
     */
    memTab[COEFF].size = params->filterLen * sizeof(Int);
    memTab[COEFF].alignment = 0;
    memTab[COEFF].space = IALG_DARAM1;
    memTab[COEFF].attrs = IALG_PERSIST;
    return (NUMBUFS);
}
/*
 *  ======== FIG_TI_free ========
 */
Int FIG_TI_free(IALG_Handle handle, IALG_MemRec memTab[])
{
    FIG_TI_Obj *fig = (Void *)handle;
    FIG_TI_alloc(NULL, NULL, memTab);
    memTab[COEFF].base = fig->coeff;
    memTab[COEFF].size = fig->filterLen * sizeof (Int);

    return (NUMBUFS);
}
/*
 *  ======== FIG_TI_initObj ========
 */
Int FIG_TI_initObj(IALG_Handle handle,
                   const IALG_MemRec memTab[], IALG_Handle parent,
                   const IALG_Params *algParams)
{
    FIG_TI_Obj *fig = (Void *)handle;
    const IFIG_Params *params = (Void *)algParams;

    if (params == NULL) {
       params = &IFIG_PARAMS;  /* use defaults if algParams == NULL */
    }
    /* initialize the FIG object's fields */
    fig->coeff = memTab[COEFF].base;
   fig->filterLen = params->filterLen;
   /* copy coefficients into on-chip persistant memory */
  memcpy((Void *)fig->coeff,
       (Void *)params->coeffPtr, params->filterLen * sizeof (Int));
    return (IALG_EOK);
}
/*
 *  ======== FIG_TI_getStatus ========
 */
Void FIG_TI_getStatus(IFIG_Handle handle, IFIG_Status *status)
{
   FIG_TI_Obj *fig = (Void *)handle;
  status->coeffPtr = fig->coeff;
}
/*
*  ======== FIG_TI_moved ========
 */
Void FIG_TI_moved(IALG_Handle handle,
                 const IALG_MemRec memTab[], IALG_Handle parent,
                   const IALG_Params *algParams)
{
  FIG_TI_Obj *fig = (Void *)handle;
   /* initialize the FIG object's fields */
   fig->coeff = memTab[COEFF].base;
}
```

A.16 fig_ti.h - Vendor-Specific Filter Group Interface

Text

```
/*
*  ======== fig_ti.h ========
*  Vendor specific (TI) interface header for Filter Group algorithm
*/
#ifndef FIG_TI_
#define FIG_TI_
#include <ialg.h>
#include <ifig.h>
/*
 *  ======== FIG_TI_exit ========
 *  Required module finalization function
 */
extern Void FIG_TI_exit(Void);
/*
 *  ======== FIG_TI_init ========
 *  Required module initialization function
 */
extern Void FIG_TI_init(Void);
/*
 *  ======== FIG_TI_IALG ========
 *  TI's implementation of FIG's IALG interface
 */
extern IALG_Fxns FIG_TI_IALG;
/*
 *  ======== FIG_TI_IFIG ========
 *  TI's implementation of FIG's IFIG interface
 */
extern IFIG_Fxns FIG_TI_IFIG;
#endif  /* FIG_TI_ */
```

A.17 fig_ti_priv.h - Private Vendor-Specific Filter Group Header

Text

```
/*
 *  ======== fig_ti_ifigvt.c ========
 *  This file contains the function table definitions for all interfaces
 *  implemented by the FIG_TI module.
 */
#include <std.h>
#include <ialg.h>
#include <ifig.h>
#include <fig_ti.h>
#include <fig_ti_priv.h>
#define IALGFXNS \
   &FIG_TI_IALG,   /* implementation ID */                         \
   NULL,           /* activate (NULL => nothing to do) */          \
    FIG_TI_alloc,   /* alloc */                                     \
    NULL,           /* control (NULL => no control operations) */   \
    NULL,           /* deactivate (NULL => nothing to do) */        \
    FIG_TI_free,    /* free */                                      \
   FIG_TI_initObj, /* init */                                      \
    FIG_TI_moved,   /* moved */                                     \
    NULL            /* numAlloc() (NULL => IALG_MAXMEMRECS) */        \

/*
 *  ======== FIG_TI_IFIG ========
 */
IFIG_Fxns FIG_TI_IFIG = {       /* module_vendor_interface */
   IALGFXNS,               /* IALG functions */
    FIG_TI_getStatus        /* IFIG getStatus */
};
/*
 *  ======== FIG_TI_IALG ========
 *  This structure defines TI's implementation of the IALG interface
 *  for the FIG_TI module.
 */
#ifdef _TI_
asm("_FIG_TI_IALG .set _FIG_TI_IFIG");
#else
/*
 *  We duplicate the structure here to allow this code to be compiled and
 *  run non-DSP platforms at the expense of unnecessary data space
*  consumed by the definition below.
*/
IALG_Fxns FIG_TI_IALG = {       /* module_vendor_interface */
    IALGFXNS,               /* IALG functions */
};
#endif
```

A.18 figtest.c- Example Client of FIG and ALG

Text

```
/* *  ======== figtest.c ========
*  Example use of FIG, FIR and ALG modules.  This test creates some
 *  number of FIR filters that all share a common set of coefficients
 *  and working buffer.  It then applies the filter to the data and
 *  displays the results.
 */
#include <std.h>
#include <fig.h>
#include <fir.h>
#include <log.h>
#include <fig_ti.h>
#include <fir_ti.h>
extern LOG_Obj trace;
#define NUMFRAMES   2           /* number of frames of data to process */
#define NUMINST     4           /* number of FIR filters to create */
#define FRAMELEN    7           /* length of in/out frames (words) */
#define FILTERLEN   8           /* length of coeff array (words) */
Int coeff[FILTERLEN] = {        /* filter coefficients */
    1, 2, 3, 4, 4, 3, 2, 1
};
Int in[NUMINST][FRAMELEN] = {   /* input data frames */
    {1, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 0}
};
Int out[NUMINST][FRAMELEN];     /* output data frames */
static Void display(Int a[], Int n);
/*
*  ======== main ========
*/
Int main(Int argc, String argv[])
{
    FIG_Params figParams;
    FIR_Params firParams;
    FIG_Status figStatus;
    FIG_Handle group;
    FIR_Handle inst[NUMINST];
    Bool status;
    Int i, n;

   FIG_init();
   FIR_init();
   figParams = FIG_PARAMS;
   figParams.filterLen = FILTERLEN;
   figParams.coeffPtr = coeff;
  /* create the filter group */
 if ((group = FIG_create(&FIG_TI_IFIG, &figParams)) != NULL) {
     /* get FIG pointers */
      FIG_getStatus(group, &figStatus);
    /* create multiple filter instance objects that reference group */
   firParams = FIR_PARAMS;
   firParams.frameLen = FRAMELEN;
    firParams.filterLen = FILTERLEN;
  firParams.coeffPtr = figStatus.coeffPtr;
   for (status = TRUE, i = 0; i < NUMINST; i++) {
         inst[i] = FIR_create(&FIR_TI_IFIR, &firParams);
     if (inst[i] == NULL) {
             status = FALSE;
     }
     }
     /* if object creation succeeded, apply filters to data */
  if (status) {
```

```
          /* activate group object */
             FIG_activate(group);
            /* apply all filters on all frames */
             for (n = 0; n < NUMFRAMES; n++) {
                for (i = 0; i < NUMINST; i++) {
                       FIR_apply(inst[i], in[i], out[i]);
                     display(out[i], FRAMELEN);
        }
               }
               /* deactivate group object */
              FIG_deactivate(group);
         }
          /* delete filter instances */
          for (i = 0; i < NUMINST; i++) {
          FIR_delete(inst[i]);
         }
           /* delete filter group object */
         FIG_delete(group);
        }
       FIG_exit();
        FIR_exit();
        return (0);
      }
      /*
      *  ======== display ========
       */
      static Void display(Int a[], Int n)
      {
          Int i;
         for (i = 0; i < n; i++) {
            LOG_printf(&trace, "%d ", a[i]);
          }
          LOG_printf(&trace, "\n");
      }
```

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
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265