TMS320 DSP Algorithm Standard API Reference

Literature Number: SPRU360B
December 2000

PRINTED WITH SOY INK

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

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

Intended Audience

This document assumes that you are fluent in the C language, have a good working knowledge of digital signal processing (DSP) 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 IDMA interface.
- **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 (literature number SPRU352)*
2) *TMS320 DSP Algorithm API Reference* (this document)

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`.

Contents

Figures

Chapter 1

Abstract Algorithm Interfaces

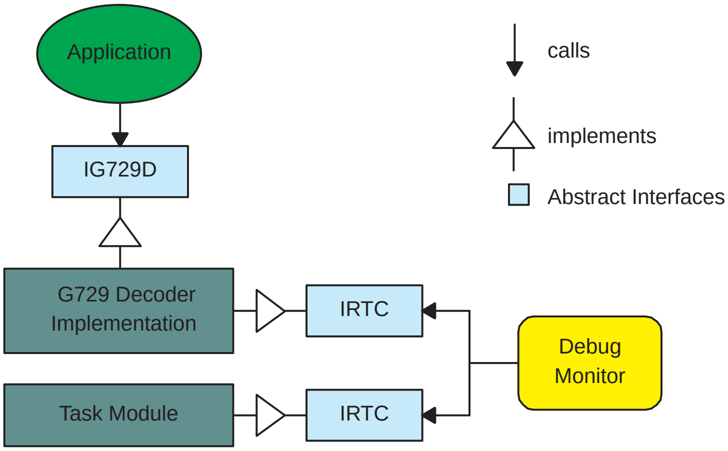
This chapter describes all of the abstract interfaces that are defined by the XDAIS specification that apply to all algorithms.

- IALG – algorithm interface defines a framework independent interface for the creation of algorithm instance objects
- IDMA – algorithm interface defining a uniform way to handle the DMA resource.
- IRTC – module trace and debug interface defines a uniform interface for enabling, disabling, and configuring the trace modes of a module.

All XDAIS algorithms **must** implement the IALG interface. XDAIS algorithms that **want** to utilize the DMA resource must implement the IDMA interface. XDAIS algorithms **should** implement the IRTC interface.

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.

Figure 1–1. Multiple Interface Support



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.

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`.

Name **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_EFAIL  –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 independant 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 independant blocks required */
    IALG_DARAM2 = 4,       /* block 2, if a 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 {
    Int       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 a this
 *  as their first element.
 */
typedef struct IALG_Params {
    Int size;    /* number of MAU in the structure */
} IALG_Params;
/*
 *  ======== IALG_Status ========
 *  Pointer to algorithm specific status structure
 *
 *  All XDAS algorithm parameter structures *must* have a this
 *  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 XDAS 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
 *               then 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 the 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.

The functions defined in IALG_Fxns fall into several categories.

1) Instance object creation, initialization, and deletion

2) Algorithmic processing

3) Instance object control and relocation

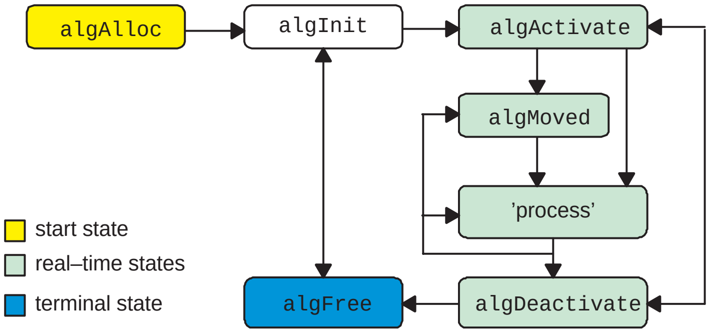
Instance object creation is complicated by removing memory allocation from the algorithm. In order for an algorithm to be used in a variety of applications, decisions about memory overlays and preemption must be made by the client rather than the algorithm. Thus, it is important to give the client as 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.

Once an algorithm instance object is created, it can be used to process data in real-time. The subclasses 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.

GThe 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 = ...;           /* initialize selected parameters */
   fxns–>algAlloc(&stdParams, ...); /* pass parameters to algorithm */

   tiParams = FIR_TI_PARAMS;        /* initialize all fields to defaults */
   tiParams.coeff = ...;            /* initialize selected parameters */
   fxns–>algAlloc(&tiParams, ...);  /* pass parameters to algorithm */
}
Int FIR_TI_algAlloc(IALG_Params *clientParams, ...)
{
   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.

This same technique is used to extend the algorithm status structures. In this case, however, all algorithm status structures start with the `IALG_Status` 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 ALG_Obj {
   IALG_Fxns    fxns;        /* algorithm functions */
} ALG_Obj;
```

```
IALG_Handle ALG_create(IALG_Fxns *fxns, IALG_Params *params)
{
    IALG_MemRec   memTab[MAXMEMRECS];
    IALG_Handle   alg = NULL;
    Int           n;

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

    return (alg);
}
Void ALG_delete(IALG_Handle alg)
{
    IALG_MemRec memTab[MAXMEMRECS];
    Int n;

    n = alg->fxns->algFree(alg, memTab);
    freeMemory(memTab, n);
}
```

In order to implement the IALG interface, all algorithm objects *must* be defined with `IALG_Obj` as their first field. This insures that all pointers to algorithm objects can be treated as pointers to `IALG_Obj` structures.

The framework functions outlined above are just examples of how to use the IALG functions to create a simple object create and delete function. Other frameworks might create objects very differently. For example, one can imagine a framework that creates multiple objects at the same time by first invoking the `algAlloc()` function for all objects, optimally allocating memory for the entire collection of objects, and then completing the initialization of the objects. By considering the memory requirements of all objects prior to allocation, such a framework can more optimally assign memory to the required algorithms.

Once an algorithm object instance is created it can be used to process data. However, it is important that if the algorithm defines `algActivate()` and `algDeactivate()` methods, then these must bracket the execution of any of the algorithm's processing functions. The following function could be used, for example, to execute any implementation of the IFIR interface on a set of buffers.

```
Void FIR_apply(FIR_Handle alg, Int *in[], Int *out[])
{
    /* do app specific initialization of scratch memory */
    if (alg->fxns->ialg.algActivate != NULL) {
        alg->fxns->ialg.algActivate(alg);
    }
```

```
    /* filter data */
    alg–>fxns–>filter(alg, in, out);

    /* do app specific store of persistent data */
    if (alg–>fxns–>ialg.algDeactivate != NULL) {
        alg–>fxns–>ialg.algDeactivate(alg);
    }
}
```

This implementation of `FIR_apply()` assumes that all persistent memory is not shared; thus, it does not restore this data prior to calling `algActivate()` and it does not save this memory after `algDeactivate()`. If a framework shares persistent data among algorithms, it must insure that this data is properly restored prior to running any processing methods of the algorithms.

If an algorithm's processing functions are always executed as shown in the `FIR_apply()` function above, there is no need for the `algActivate()` and `algDeactiveate()` functions. To save the overhead of making two function calls, their functionality would be folded into the processing functions. The purpose of `algActivate()` and `algDeactivate()` is to enable the algorithm's processing functions to be called multiple times between calls to `algActivate()` and `algDeactivate()`. This allows the algorithm writer the option of factoring data initialization functions, such as initialization of scratch memory, into the `algActivate()` function. The overhead of this data movement can then be amortized across multiple calls to processing functions.

See the example of a simple `FIR_TI` filter module that implements the IALG interface (in Appendix A). The `algActivate()` function copies filter history and coefficients into scratch DARAM, the `algDeactivate()` function copies history data to external persistent memory, and the filter function treats the filter coefficient and history memory as persistent data. In this example, the filter function can optimally process the data by minimizing the per frame overhead associated with saving and restoring persistent data.

Name **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 IDMA 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()`

Name **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 optional output parameter. `algAlloc()` may return a pointer to another set of ILAG functions to the client. If this output value is set to a non-`NULL` value, the client creates an instance object using this set of IALG functions. The resulting 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.

- ❑ Exactly n elements of the `memTab[]` array are initialized, where n is the return value from this operation. The base field of each element is not initialized, however.
- ❑ 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.

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_Obj  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 = &ENCODERATTRS; /* use default parameters */
   }
   memTab[0].size = sizeof (EncoderObj);
   memTab[0].alignment = 1;
   memTab[0].type = IALG_PERSIST;
   memTab[0].space = IALG_EXTERNAL;

   memTab[1].size = params–>frameDuration * 8 * sizeof(int);
   memTab[1].alignment = 1;      /* no alignment */
   memTab[1].type = IALG_PERSIST;
   memTab[1].space = IALG_DARAM; /* dual-access on-chip */
   return (2);
}
```

See Also `algFree()`

Name **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, …} EncoderCmd;

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

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

See Also `algInit()`

Name 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()`

Name **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.

Name **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. Thus, the algorithm is required to properly operate (although much less efficiently) even if it is not given memory in, say on-chip DARAM.

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.
- ❑ The buffer pointed to in `memTab[0]` is initialized to contain all 0s.
- ❑ `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()` 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.

Postconditions The following conditions is true immediately after returning from this method.

- With the exception of any initialization performed by `algActivate()`, all of the instance's persistent memory is initialized and the object is ready to be used.
- If the algorithm has implemented the IDMA 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 = &ENCODERATTRS;  /* use default parameters */
   }

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

   return (IALG_EOK);
}
```

See Also `algAlloc(), algMoved()`

Name **algMoved – notify algorithm 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()`.

- 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()`

Name **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()`.

Example The example below shows how an algorithm can use another algorithm.

```
#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(..., memTab);
   return (n + NUMBUF);
}
```

See Also `algAlloc()`

Name IDMA – algorithm DMA interface

Synopsis `#include <idma.h>`

Interface

```
/*
 *  ======== idma.h ========
 */
#ifndef IDMA_
#define IDMA_

#ifdef __cplusplus
extern "C" {
#endif

#include <ialg.h>

/*
 *  ======== IDMA_Handle ========
 *  Handle to "logical" DMA channel.
 */
typedef struct IDMA_Obj *IDMA_Handle;

/*
 *  ======== IDMA_ElementSize ========
 *  8, 16 or 32-bit aligned transfer.
 */
typedef enum IDMA_ElementSize {
    IDMA_ELEM8,            /* 8-bit data element */
    IDMA_ELEM16,           /* 16-bit data element */
    IDMA_ELEM32            /* 32-bit data element */
} IDMA_ElementSize;

/*
 *  ======== IDMA_TransferType ========
 *  Type of the DMA transfer.
 */
typedef enum IDMA_TransferType {
    IDMA_1D1D,             /* 1-dimensional to 1-dimensional transfer */
    IDMA_1D2D,             /* 1-dimensional to 2-dimensional transfer */
    IDMA_2D1D,             /* 2-dimensional to 1-dimensional transfer */
    IDMA_2D2D              /* 2-dimensional to 2-dimensional transfer */
} IDMA_TransferType;

/*
 *  ======== IDMA_Params ========
 *  DMA transfer specific parameters. Defines the context of a
 *  logical channel.
 */
typedef struct IDMA_Params {
    IDMA_TransferType  xType;           /* 1D1D, 1D2D, 2D1D or 2D2D */
```

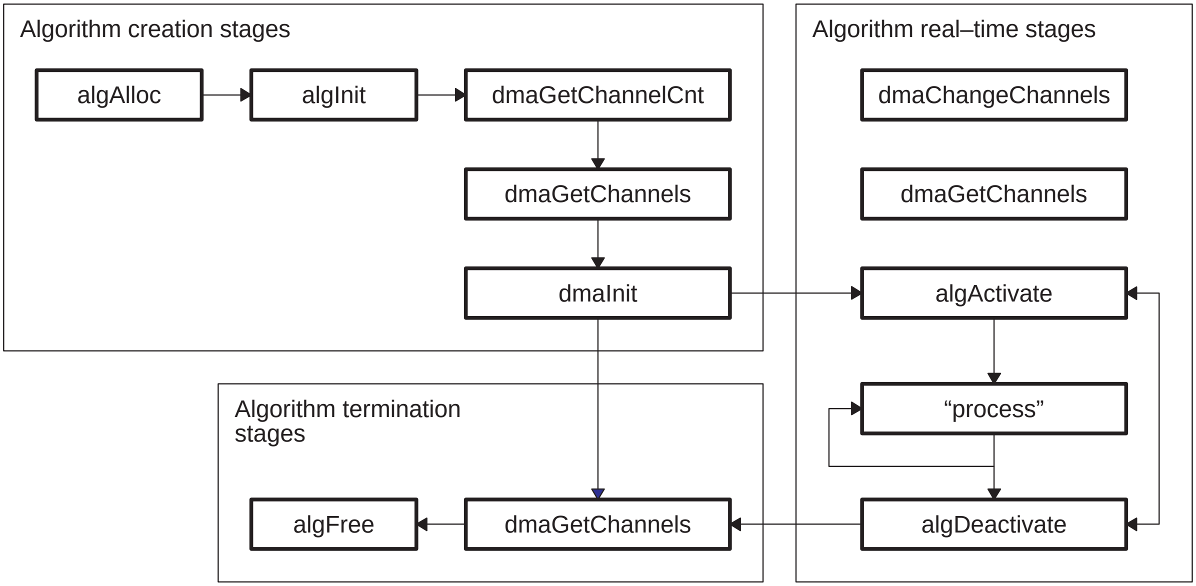
```
    IDMA_ElementSize   elemSize;        /* Element transfer size */
    Int                numFrames;       /* Num of frames for 2D transfers */
    Int                stride;          /* Jump in elemSize for 2D transfers */
} IDMA_Params;

/*
 *  ======== IDMA_ChannelRec ========
 *  DMA record used to describe the logical channels
. */
typedef struct IDMA_ChannelRec {
    Int             depth;       /* Num max (concurrent) queued transfers */
    Bool            dimensions;  /* 0 = only 1D transfers, 2D otherwise */
    IDMA_Handle  handle;         /* Handle to logical DMA channel */
} IDMA_ChannelRec;

/*
 *  ======== IDMA_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 IDMA_Fxns {
    Void     *implementationId;
    Void     (*dmaChangeChannels)(IALG_Handle, IDMA_ChannelRec *);
    Int      (*dmaGetChannelCnt)(Void);
    Int      (*dmaGetChannels)(IALG_Handle, IDMA_ChannelRec *);
    Void     (*dmaInt)(IALG_Handle, IDMA_ChannelRec *);
} IDMA_Fxns;

#ifdef ___cplusplus
}
#endif

#endif /* IDMA_ */
```

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

A module implements the IDMA interface if it defines and initializes a global structure of type `IDMA_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 `IDMA_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.

Figure 1–3. IDMA Functions Calling Sequence



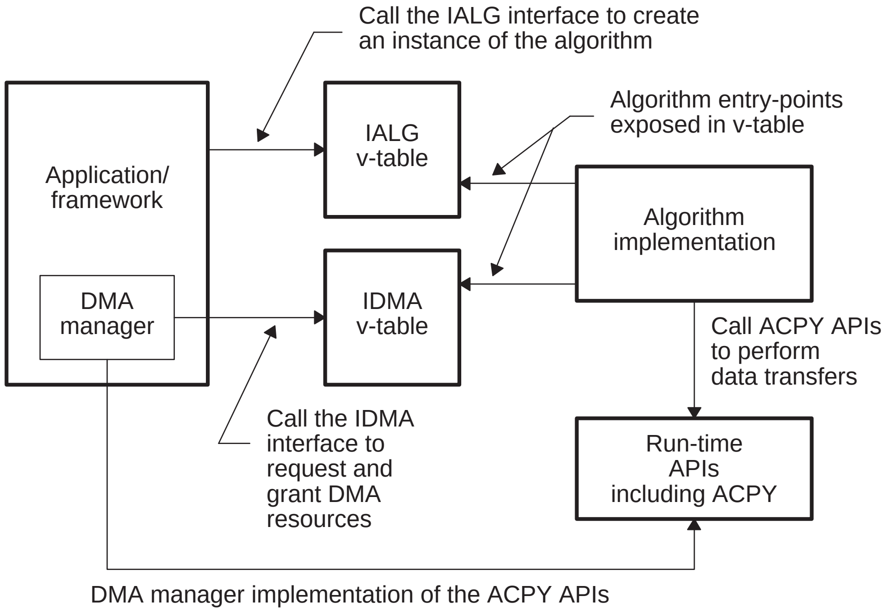
Note that the `dmaChangeChannels()` and `dmaGetChannels()` operations can be called at any time in the algorithm's real-time stages. The `algMoved()` and `algNumAlloc()` functions are omitted for simplicity.

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.

System Overview Figure 1–4 illustrates a system with an algorithm implementing the IALG and IDMA interfaces and the application with a DMA manager. Notice that the algorithm calls the ACPY runtime APIs, which is implemented by the application's DMA manager.

Figure 1–4. Algorithm Implementing the IALG and IDMA Interfaces and the Application with a DMA Manager



Name

dmaChangeChannels – notify algorithm instance that DMA resources have changed

Synopsis

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

Arguments

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

Return Value

```
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

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

Void dmaChangeChannels(IALG_Handle handle, IDMA_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()`

Name **dmaGetChannelCnt – get number of DMA resources required**

Synopsis `numRecs = dmaGetChannelCnt(Void);`

Arguments `Void`

Return Value

```
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 that `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

```
typedef struct ImageObj {
    IALG_Obj    ialg;        /* IALG object MUST be first field */
    IDMA_Handle dmaHandle_0  /* Handle for logical DMA channel 0 */
    IDMA_Handle dmaHandle_1  /* Handle for logical DMA channel 1 */
    IDMA_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()`

Name

dmaGetChannels – get algorithm object's dma requirements/holdings

Synopsis

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

Arguments

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

Return Value

```
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 `IDMA_ChannelRec` structure is undefined when this operation is called at algorithm's initialization. All fields in the `IDMA_ChannelRec` 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 the 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 or less the number returned by `dmaGetChannelCnt()`.
- ❑ `handle` must be a valid pointer to the algorithm's instance object.

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 `IDMA_ChannelRec` structure is valid only if this operation is called after `algInit()`.

Example

```
typedef struct ImageObj {
    IALG_Obj    ialg;       /* IALG object MUST be first field */
    IDMA_Handle dmaHandle_0 /* Handle for logical DMA channel 0 */
    IDMA_Handle dmaHandle_1 /* Handle for logical DMA channel 1 */
    IDMA_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, IDMA_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, IDMA_ChannelRec dmaTab[])
{
    dmaTab[0].depth = 4;       /* Max 4 concurrent transfer on this ch */
    dmaTab[0].dimensions = 1;  /* May do 2D transfer */
    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].depth = 4;       /* Max 4 concurrent transfer on this ch */
        dmaTab[1].dimensions = 1;  /* May do 2D transfer */
        dmaTab[1].handle = img->dmaHandle_1;
        dmaTab[2].depth = 4;       /* Max 4 concurrent transfer on this ch */
        dmaTab[2].dimensions = 1;  /* May do 2D transfer */
        dmaTab[2].handle = img->dmaHandle_2;
        return (3);
    }

    return (1);
}
```

See Also `dmaGetChannelCnt()`

Name **dmaInit – grant the algorithm DMA resources**

Synopsis

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

Arguments

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

Return Value

```
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.

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

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

Int dmaInit(IALG_Handle handle, IDMA_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()`

Chapter 2

ACPY Runtime APIs

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

The ACPY APIs are:

ACPY_complete()	Called by an algorithm to check if the data transfers on a specific logical channel have completed.
ACPY_configure()	Called by an algorithm to configure a logical channel.
ACPY_start()	Called by an algorithm to issue a request for a data transfer.
ACPY_wait()	Called by an algorithm to wait for all data transfers to complete on a specific logical channel.

Name **ACPY_complete – check to see if the data transfers have completed**

Synopsis `dmaDone = ACPY_complete(IDMA_Handle handle);`

Arguments `IDMA_Handle handle;  /* handle to a logical DMA channel */`

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

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

The only argument to `ACPY_complete()` specifies the logical channel used for the data transfer requested with `ACPY_start()`.

The framework implementation of `ACPY_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.

```
IDMA_Handle  dmaHandle;

if (ACPY_complete(dmaHandle) {
    startProcesingData();
{
else {
     ‘do some other work’
}

/* No more processing to do – wait for data transfers to complete */
ACPY_wait(dmaHandle);
startProcesingData();
```

See Also `ACPY_wait()`

Name **ACPY_configure – configure a logical channel**

Synopsis `Void ACPY_configure(IDMA_Handle handle, IDMA_Params *params)`

Arguments

```
IDMA_Handle handle; /* handle to logical DMA channel */
IDMA_Params params;/* Channel parameters */
```

Return Value `Void;`

Description `ACPY_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 `ACPY_configure()` specifies the logical channel subject to configuration.

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

The framework implementation of `ACPY_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.
- `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`.
- `ACPY_start()` can be called with params=NULL.

Examples

- Configure the logical channel pointed to by `handle` for a 1D1D transfer. We know that the `src` and `dst` buffers are aligned on a 32-bit boundary. Note that the `numFrames` and `stride` values will be ignored when `xType=IDMA_1D1D`.

```
IDMA_Params  params;
IDMA_Handle  dmaHandle;

params.xType = IDMA_1D1D;
params.elemSize = IDMA_ELEM32;
params.numFrames = 0;    /* Not used in 1D1D transfer */
params.stride = 0;       /* Not used in 1D1D transfer */

ACPY_configure(dmaHandle,&params);
```

- Configure the logical channel pointed to by `handle` for a 1D2D transfer. We don't 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. 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.

```
IDMA_Params  params;
IDMA_Handle  dmaHandle;

params.xType = IDMA_1D2D;
params.elemSize = IDMA_ELEM8;
params.numFrames = 8;
params.stride = 100;       /* In elemSize */

ACPY_configure(dmaHandle,&params);
```

See Also `ACPY_start()`

Name **ACPY_start – issue a request for a data transfer**

Synopsis

```
Void ACPY_start(IDMA_Handle handle,
            Void *src, Void *dst, Uns cnt, IDMA_Params *params)
```

Arguments

```
IDMA_Handle handle;  /* handle to DMA resource*/
Void* src;          /* Source address for data transfer */
Void* dst;          /* Destination addr for data transfer */
Uns cnt;             /* Number of elements in a frame */
IDMA_Params *params; /* Context parameters */
```

Description ACPY_start() issues a request for a data transfer. The implementation of ACPY_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 ACPY_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 maximum number of repetitive calls to ACPY_start() before calling ACPY_wait() or is decided by the depth field in the IDMA_ChannelRec structure for the particular logical channel pointed to by handle.

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

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

The fourth argument to ACPY_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 fifth argument to ACPY_start() is used to control the configuration of the logical channel pointed to by handle. If params=NULL, the current configuration of the logical channel will be used when transferring data from src to dst. If params!=NULL, the data transfer elements from src to dst will be configured as specified in the params structure. The logical channel pointed to by handle will contain this configuration after returning from the operation. It can be changed by ACPY_configure() or another call to ACPY_start() with params!=NULL.

The framework implementation of ACPY_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.
- If params=NULL, the logical channel must already be configured through ACPY_configure() or a previous ACPY_start() with params!=NULL.

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

- If `params!=NULL`, the channel configuration will change to the values pointed to by `params`.
- The data transfer is in progress or in a queue waiting to be started.

Examples

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

```
IDMA_Handle  dmaHandle;

ACPY_start(dmaHandle, src, dst, 100, NULL);
```

- Start a DMA transfer on a logical channel that currently has a different channel configuration. The transfer is 2D2D, 16-bits elements, 32 elements in a frame, 18 frames and 20 elements between end of a frame to the start of the next frame.

```
 IDMA_Params  params;
 IDMA_Handle  dmaHandle;

 params.xType = IDMA_2D2D;
 params.elemSize = IDMA_ELEM16;
 params.numFrames = 18;
 params.stride = 20;       /* In elemSize */

ACPY_start(dmaHandle, src, dst, 32, &params);
```

- Start three DMA transfers on the same logical channel. The channel is not pre-configured, however, all three transfers need the same configuration. The transfer is 2D2D, 8-bit elements, 16 elements in a frame, 8 frames and 64 elements between end of a frame to the start of the next frame.

```
 IDMA_Params  params;
 IDMA_Handle  dmaHandle;

 params.xType = IDMA_2D2D;
 params.elemSize = IDMA_ELEM8;
 params.numFrames = 8;
 params.stride = 64;       /* In elemSize */

ACPY_start(dmaHandle, src1, dst1, 16, &params);
ACPY_start(dmaHandle, src2, dst2, 16, NULL);
ACPY_start(dmaHandle, src3, dst3, 16, NULL);
```

See Also `ACPY_configure()`, `ACPY_complete()`, `ACPY_wait()`.

Name **ACPY_wait – wait for the data transfers to complete**

Synopsis `Void ACPY_wait(IDMA_Handle handle);`

Arguments `IDMA_Handle handle;  /* handle to DMA resource*/`

Return Value `Void`

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

The only argument to `ACPY_wait()` specifies the logical channel used for the data transfer requested with `ACPY_start()`.

The framework implementation of `ACPY_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.

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 be `handle`.

```
IDMA_Handle  dmaHandle;

ACPY_wait(dmaHandle);
```

See Also `ACPY_complete()`

Chapter 3

Supplementary APIs

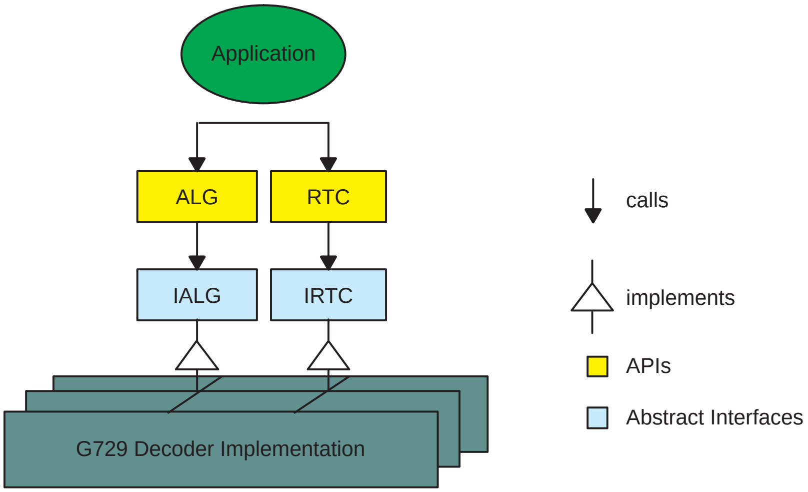
This chapter describes supplementary module APIs that are available to the clients of XDAIS algorithms but are *not* part of the core run-time support. These modules are logically part of an XDAIS framework and are provided to simplify the use and management of eXpressDSP-compliant algorithms. These APIs define a simple XDAIS run-time support library that is provided in the TMS320 DSP Algorithm Standard Developer's Kit.

- ALG – module for the creation of algorithm instance objects
- RTC – module for enabling, disabling, and configuring the trace modes of any algorithm module.

These APIs and any run-time support library provided by the TMS320 DSP Algorithm Standard Developer's Kit 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.

Name 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.

Name **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()`

Name **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()`

Name **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` is indicates that the operation completed successfully; all other return values indicate failure.

Example

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

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()`

Name — **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()`

Name **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()`

Name

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()`

Name

ALG_exit – module clean-up

Synopsis

```
Void ALG_exit(VOID);
```

Arguments

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

Return Value

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

Description `ALG_exit()` is called during system shutdown to perform any run-time finalization necessary for the algorithm module as a whole.

See Also `ALG_delete()`

Name **RTC – Generic Algorithm Trace Module**

Synopsis `#include <rtc.h>`

Interface

```
/*----------------------------*/
/*    TYPES AND CONSTANTS    */
/*----------------------------*/
#define RTC_ENTRY             IRTC_ENTRY
#define RTC_WARNING           IRTC_CLASS1

typedef struct RTC_Desc {
    IRTC_Fxns      fxns;   /* trace functions */
    IALG_Handle    handle; /* algorithm instance handle */
} RTC_Desc;

/*----------------------------*/
/*        FUNCTIONS          */
/*----------------------------*/

RTC_bind();           /* bind output log to module */
RTC_create();         /* create a trace instance object */
RTC_delete();         /* delete trace instance’s object */
RTC_disable();        /* disable all trace levels */
RTC_enable();         /* (re)enable trace levels */
RTC_exit();           /* RTC module finalization */
RTC_get();            /*get trace level */
RTC_init();           /* RTC module initialization */
RTC_set();            /* set trace level */
```

Description The RTC module provides a generic (universal) interface used to control the trace capabilities of any algorithm instance. The functions provided by this module use the IRTC interface functions to dynamically control the various trace levels supported by algorithm objects. Any module that implements the IRTC and the IALG interface can be used by RTC.

Name **RTC_bind – bind an output log to a module**

Synopsis

```
RTC_bind(IRTC_Fxns *fxns, LOG_Obj *log);
```

Arguments

```
IRTC_Fxns   *fxns;   /* IRTC interface functions */
LOG_Obj     *log;    /* log pointer for trace output */
```

Return Value `Void`

Description `RTC_bind()` sets the output log of a module. This operation is typically called during system initialization and it must not preempt any other operation supported by the implementing module.

The `IRTC_Fxns` pointer must be a pointer to a module's implementation of the IRTC interface. The second argument must be a valid pointer to a DSP/BIOS LOG object.

See Also `RTC_create()`

Name **RTC_create – initialize a trace descriptor**

Synopsis

```
rtc = RTC_create(RTC_Desc *rtc,
              ALG_Handle alg, IRTC_Fxns *fxns);
```

Arguments

```
RTC_Desc   *rtc;   /* trace object pointer */
ALG_Handle alg;    /* algorithm instance object pointer */
IRTC_Fxns  *fxns;  /* IRTC interface functions */
```

Return Value

```
RTC_Desc   *rtc;   /* trace object pointer */
```

Description RTC_create() initializes the trace descriptor structure. The first argument to RTC_create() is a pointer to a trace descriptor structure. This structure is initialized using the algorithm object and a pointer to a module's IRTC implementation functions. The algorithm object structure pointer must be a pointer to a previously created algorithm object (via ALG_create()).

The IRTC_Fxns pointer must be a pointer to a module's implementation of the IRTC interface and this module must also implement the IALG interface used to construct the ALG_Obj structure. RTC_create performs a run-time check to insure these two interface implementations are consistent.

If RTC_create() returns NULL then it failed; otherwise, it returns its first argument.

See Also ALG_create()

Name **RTC_delete – delete a trace instances descriptor**

Synopsis

```
Void RTC_delete(RTC_Desc *rtc);
```

Arguments

```
RTC_Desc   *rtc;   /* rtc object descriptor */
```

Return Value

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

Description `RTC_delete()` deletes the trace descriptor, rtc, which was initialized by RTC_create(). If rtc is NULL, RTC_delete() simply returns.

See Also `RTC_create()`

Name **RTC_disable – disable all trace levels**

Synopsis

```
Void RTC_disable(constRTC_Desc *rtc);
```

Arguments

```
RTC_Desc   *rtc;   /* rtc object descriptor */
```

Return Value

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

Description `RTC_disable()` sets the current trace bit mask for the instance object to a value such that no diagnostic information is produced in real-time by the trace object. The first argument to RTC_disable() is a trace descriptor initialized via RTC_create(). If rtc is NULL, RTC_disable simply returns.

See Also `RTC_create()`

Name **RTC_enable – enable trace at last set level**

Synopsis

```
Void RTC_enable(constRTC_Desc *rtc);
```

Arguments

```
RTC_Desc   *rtc;   /* rtc object descriptor */
```

Return Value

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

Description `RTC_enable()` sets the current trace bit mask for the instance object to the last set value representing the level of trace and diagnostic information that should be produced in real-time by the trace object.

The first argument to RTC_enable() is a trace descriptor initialized via RTC_create(). IF rtc is NULL, RTC_enable simply returns.

See Also `RTC_create(), RTC_set()`

Name **RTC_exit – module clean-up**

Synopsis `Void RTC_exit(Void);`

Arguments `Void /* none */`

Return Value `Void /* none */`

Description `RTC_exit()` runs during system shutdown to perform any run-time finalization necessary for the RTC module as a whole.

See Also `ALG_exit()`

Name RTC_get – get the current trace mask setting

Synopsis `mask = RTC_get(RTC_Desc *desc);`

Arguments

```
RTC_Desc     *desc;     /* trace instance descriptor */
```

Return Value

```
RTC_Mask     mask;      /* current trace mask */
```

Description `RTC_get()` returns the current setting of the trace mask for a trace descriptor. The first (and only) argument to `RTC_get()` is a trace descriptor initialized via `RTC_create()`.

Example

```
main()
{
   RTC_Desc trace;
   ALG_Handle alg;

   alg = …;
   RTC_create(&desc, alg, &FIR_TI_IRTC);

   /* get current trace mask for alg */
   mask = RTC_get(&desc);
   …
}
```

See Also `RTC_create()`

Name **RTC_init – module initialization**

Synopsis

```
Void RTC_init(Void);
```

Arguments

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

Return Value

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

Description `RTC_init()` is called during system startup to perform any run-time initialization necessary for the RTC module as a whole.

See Also `RTC_create()`

Name **RTC_set – set the current trace mask setting**

Synopsis

```
RTC_set(RTC_Desc *desc, RTC_Mask mask);
```

Arguments

```
RTC_Desc      *desc;    /* trace instance descriptor */
RTC_Mask      mask;     /* new trace mask */
```

Return Value

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

Description `RTC_set()` sets the current trace mask for an algorithm instance specified by the descriptor pointer desc.

The first argument to `RTC_set()` is a trace descriptor initialized via `RTC_create()`. The second argument is a bit mask representing the level of trace and diagnostic information that should be produced in real-time by the trace object.

Example

```
main()
{
   RTC_Desc desc;
   ALG_Handle alg;

   alg = …;
   RTC_create(&desc, alg, &FIR_TI_IRTC);

   /* set current trace mask for alg to RTC_ENTER */
   RTC_set(&desc, RTC_ENTER);
   …
}
```

See Also `RTC_create()`

Appendix A

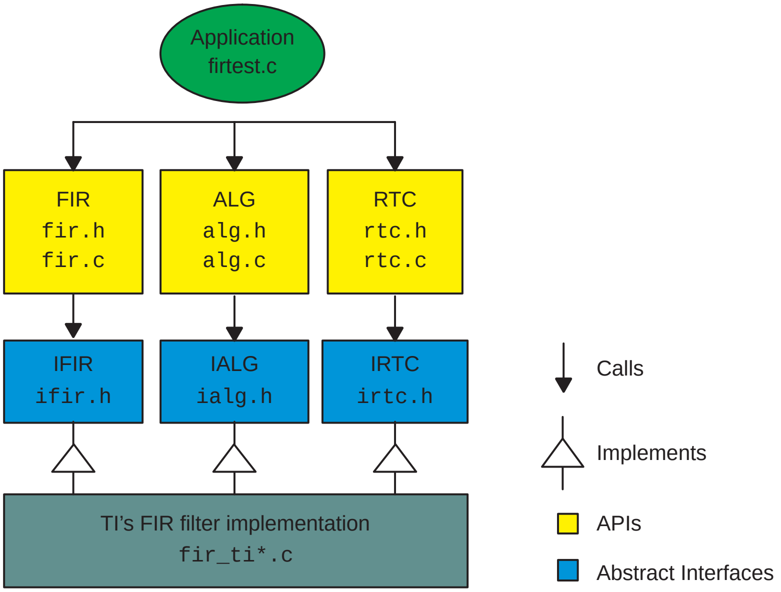
Example Algorithm Implementation

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`). 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.

Although this example is eXpressDSP-compliant, it has two significant shortcomings: all of the IALG interface functions are implemented in a single file and the algorithm is written in C. By implementing all IALG functions in a single file, it is not possible to use the process function without linking all of the other IALG functions into the application. Figure A–1 illustrates the relationship between these files.

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
fir_ti_irtc_o54	58	0	0	0	6	0

Name **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_ */
```

Name **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_ */
```

Name **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()
{
}
```

Name **fir_ti.c – Vender Specific FIR Module Implementation**

Text

```
/*
 *  ======== fir_ti_ialg.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));
}
```

```
/*
 *  ======== FIR_TI_free ========
 */
Int FIR_TI_free(IALG_Handle handle, IALG_MemRec memTab[])
{
    FIR_TI_Obj *fir = (Void *)handle;

    FIR_TI_alloc(NULL, NULL, memTab);

    memTab[HISTORY].base = fir->history;
    memTab[HISTORY].size = fir->filterLenM1 * sizeof(Int);

    memTab[WORKBUF].size =
        (fir->filterLenM1 + fir->frameLen) * sizeof(Int);
    memTab[WORKBUF].base = fir->workBuf;

    return (NUMBUFS);
}
/*
 *  ======== FIR_TI_initObj ========
 */
Int FIR_TI_initObj(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle p,
                const IALG_Params *algParams)
{
    FIR_TI_Obj *fir = (Void *)handle;
    const IFIR_Params *params = (Void *)algParams;

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

    fir->coeff = params->coeffPtr;
    fir->workBuf = memTab[WORKBUF].base;
    fir->history = memTab[HISTORY].base;
    fir->filterLenM1 = params->filterLen - 1;
    fir->frameLen = params->frameLen;

    return (IALG_EOK);
}
```

```
/*
 *  ======== FIR_TI_moved ========
 */
Void FIR_TI_moved(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle p,
                const IALG_Params *algParams)
{
    FIR_TI_Obj *fir = (Void *)handle;
    const IFIR_Params *params = (Void *)algParams;

    if (params != NULL) {
        fir->coeff = params->coeffPtr;
    }

    fir->workBuf = memTab[WORKBUF].base;
    fir->history = memTab[HISTORY].base;
}
```

Name 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
 *  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 <irtc.h>
#include <itst.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;
```

```
/*
 *  ======== FIR_TI_IRTC ========
 *  TI's implementation of FIR's IRTC interface
 */
extern IRTC_Fxns FIR_TI_IRTC;

/*
 *  ======== 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_ */
```

Name **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 <irtc.h>
#include <itst.h>
#include <ifir.h>
#include <log.h>

typedef struct FIR_TI_Obj {
    IALG_Obj    alg;            /* MUST be first field of XDAIS algs */
    IRTC_Mask   mask;           /* current test/diag mask setting */
    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 LOG_Obj *FIR_TI_rtcOut;   /* our output trace log */
```

```
/*
 *  ======== FIR_TI_trace ========
 *  Our equivalent of ”printf”
 */
 #define FIR_TI_trace(f, a1, a2)                    \
    if (FIR_TI_rtcOut != NULL) {                    \
        LOG_printf(FIR_TI_rtcOut, (f), (a1), (a2)); \
    }

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[]);

extern IRTC_Mask FIR_TI_rtcGet(IRTC_Handle handle);

extern Void FIR_TI_rtcBind(LOG_Obj *log);

extern Void FIR_TI_rtcSet(IRTC_Handle handle, IRTC_Mask mask);

#endif  /* FIR_TI_PRIV_ */
```

Name **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);
}
```

Name **fir_ti_irtc.c – Vendor Specific Implementation of IRTC Interface**

Text

```
/*
 *  ======== fir_ti_irtc.c ========
 *  Filter Module IRTC implementation – TI’s implementation of the
 *  IRTC interface for the FIR filter algorithm
 */
#include <std.h>

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


/*
 *  ======== FIR_TI_rtcOut ========
 *  This module’s output trace log.
 */
LOG_Obj *FIR_TI_rtcOut = NULL;

/*
 *  ======== FIR_TI_rtcBind ========
 */
Void FIR_TI_rtcBind(LOG_Obj *log)
{
    FIR_TI_rtcOut = log;

    FIR_TI_trace(”FIR_TI_rtcBind(0x%lx)\n”, log, NULL);
}


/*
 *  ======== FIR_TI_rtcGet ========
 */
IRTC_Mask FIR_TI_rtcGet(IRTC_Handle handle)
{
    FIR_TI_Obj *fir = (Void *)handle;

    FIR_TI_trace(”FIR_TI_rtcGet(0x%lx) = 0x%x\n”, handle, fir–>mask);

    return (fir–>mask);
}
```

```
/*
 *  ======== FIR_TI_rtcSet ========
 */
Void FIR_TI_rtcSet(IRTC_Handle handle, IRTC_Mask mask)
{
    FIR_TI_Obj *fir = (Void *)handle;

    FIR_TI_trace(”FIR_TI_rtcSet(0x%lx, 0x%x)\n”, handle, mask);

    fir–>mask = mask;
}
```

Name **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
```

Name

fir_ti_irtcvt.c – Vendor Specific IRTC Function Table

Text

```
/*
 *  ======== fir_ti_irtcvt.c ========
 *  This file contains the function table definitions for the
 *  IRTC interface implemented by the FIR_TI module.
 *
 *  We place these tables in a separate file for two reasons:
 *     1. We want 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 <irtc.h>

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


/*
 *  ======== FIR_TI_IRTC ========
 *  This structure defines TI’s implementation of the IRTC interface
 *  for the FIR_TI module.
 */
IRTC_Fxns FIR_TI_IRTC = {
    &FIR_TI_IALG,       /* module ID */
    FIR_TI_rtcBind,     /* rtcBind */
    FIR_TI_rtcGet,      /* rtcGet */
    FIR_TI_rtcSet       /* rtcSet */
};
```

Name **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”);
}
```

Name **firtest1.c – example client of ALG, RTC, and FIR**

Text

```
/*
 *  ======== firtest1.c ========
 *  This example shows how the trace interface (if implemented)
 *  can be used by an application.  It also 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 <rtc.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;
    RTC_Desc rtc;

    ALG_init();
    FIR_init();
    RTC_init();

    /* bind output log to FIR_TI module */
    RTC_bind(&FIR_TI_IRTC, &trace);

    /* create an instance of a FIR algorithm */
    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 && RTC_create(&rtc, alg, &FIR_TI_IRTC) != NULL) {

        RTC_set(&rtc, RTC_ENTER);                   /* enable trace */
        FIR_apply((FIR_Handle)alg, input, output);  /* filter data */
        display(output, FRAMELEN);                  /* display result */

        RTC_delete(&rtc);       /* delete rtc descriptor */
        ALG_delete(alg);        /* delete alg instance */
    }

    RTC_exit();
    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”);
}
```

Name | **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_ */
```

Name **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_ */
```

Name 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)
{
}
```

Name **fig_ti.c – Vendor-Specific Filter Group 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;
}
```

Name **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_ */
```

Name **fig_ti_ifigvt.h – Vendor-Specific FIG Function Table**

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
```

Name **fig_ti_priv.h – Private Vendor-Specific Filter Group Header**

Text

```
/*
 *  ======== fig_ti_priv.h ========
 *  Internal vendor specific (TI) interface header for FIG
 *  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 FIG algorithm.
 */
#ifndef FIG_TI_PRIV
#define FIG_TI_PRIV

#include <ialg.h>

typedef struct FIG_TI_Obj {
    IALG_Obj    alg;            /* MUST be first field of XDAIS algs */
    Int         *coeff;         /* on-chip persistant coeffient array */
    Int         filterLen;      /* filter length (in words) */
} FIG_TI_Obj;

extern Int FIG_TI_alloc(const IALG_Params *,IALG_Fxns **, IALG_MemRec *);

extern Int FIG_TI_free(IALG_Handle, IALG_MemRec *);

extern Void FIG_TI_getStatus(IFIG_Handle handle, IFIG_Status *status);

extern Int FIG_TI_initObj(IALG_Handle,
                  const IALG_MemRec *, IALG_Handle, const IALG_Params *);

extern Void FIG_TI_moved(IALG_Handle,
                  const IALG_MemRec *, IALG_Handle, const IALG_Params *);

#endif
```

Name

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");
}
```

Appendix B

Glossary

A

Abstract Interface: An interface defined by a C header whose functions are specified by a structure of function pointers. By convention these interface headers begin with the letter “i” and the interface name begins with “I”. Such an interface is abstract because, in general, many modules in a system implement the same abstract interface; i.e., the interface defines abstract operations supported by many modules.

Algorithm: Technically, an algorithm is a sequence of operations, each chosen from a finite set of well-defined operations (for example, computer instructions), that halts in a finite time, and computes a mathematical function. In this specification, however, we allow algorithms to employ heuristics and do not require that they always produce a correct answer.

API: Acronym for application programming interface. A specific set of constants, types, variables, and functions used to programmatically interact with a piece of software.

C

Client: The term client denotes any piece of software that uses a function, module, or interface. For example, if the function `a()` calls the `function b()`, `a()` is a client of `b()`. Similarly, if an application `App` uses module `MOD`, `App` is a client of `MOD`.

Concrete Interface: An interface defined by a C header whose functions are implemented by a single module within a system. This is in contrast to an abstract interface where multiple modules in a system can implement the same abstract interface. The header for every module defines a concrete interface.

Critical Section: A critical section of code is one in which data that can be accessed by other threads are inconsistent. At a higher level, a critical section is a section of code in which a guarantee you make to other threads about the state of some data may not be true.

If other threads can access these data during a critical section, your program may not behave correctly. This may cause it to crash, lock up, or produce incorrect results.

In order to insure proper system operation, other threads are denied access to inconsistent data during a critical section (usually through the use of locks). Poor system performance could be the result if some of your critical sections are too long.

E

Endian: Refers to which bytes are most significant in multi-byte data types. In big-endian architectures, the leftmost bytes (those with a lower address) are most significant. In little-endian architectures, the rightmost bytes are most significant.

HP, IBM, Motorola 68000, and SPARC systems store multi-byte values in big-endian order, while Intel 80x86, DEC VAX, and DEC Alpha systems store them in little-endian order. Internet standard byte ordering is also big-endian. The TMS320C6000 is bi-endian because it supports both systems.

F

Frame: Algorithms often process multiple samples of data at a time, referred to as a frame. In addition to improving performance, some algorithms require specific minimum frame sizes to operate properly.

Framework: Part of an application that is designed to remain invariant while selected software components are added, removed, or modified. Very general frameworks are sometimes described as application-specific operating systems.

I

Instance: The specific data allocated in an application that defines a particular object.

Interface: A set of related functions, types, constants, and variables. An interface is often specified with a C header file.

Interrupt Latency: The maximum time between when an interrupt occurs and its corresponding interrupt service routine (ISR) starts executing.

M

Method: A synonym for a function that is part of an interface.

Module: A module is an implementation of one (or more) interfaces. In addition, all modules follow certain design elements that are common to *all* XDAIS compatible software components. Roughly speaking, a module is a C language implementation of a C++ class.

Multithreading: Multithreading is the management of logically concurrent threads within the same program or system. Most operating systems and modern computer languages also support multithreading.

P

Preemptive: A property of a scheduler that allows one task to asynchronously interrupt the execution of the currently executing task and switch to another task. The interrupted task is *not* required to call any scheduler functions to enable the switch.

R

Reentrant: A property of a program or a part of a program in its executable version, that can be entered repeatedly, or can be entered before previous executions have been completed. Each execution of such a program is independent of all other executions.

S

Scratch Memory: Memory that can be overwritten without loss; i.e., prior contents need not be saved and restored after each use.

Scratch Register: A register that can be overwritten without loss; i.e., prior contents need not be saved and restored after each use.

T

Thread: The program state managed by the operating system that defines a logically independent sequence of program instructions. This state may be as small as the program counter (PC) value but often includes a large portion of the CPUs register set.

Index