Application Report
SPRA810 - January 2002

A Consumer's Guide to Using eXpressDSP-Compliant Algorithms

Randy Wu *DSP Software Technical Staff*

ABSTRACT

This document, along with the latest TMS320™ DSP Algorithm Standard (referred to as "XDAIS" throughout the document) API Reference Manual (SPRU360), will assist the XDAIS algorithm consumer to write the necessary source code to integrate a XDAIS-compliant algorithm into an existing DSP system application.

The procedure outlined in this document is ideal for consumers who already have an existing application framework in place and need to quickly integrate the XDAIS algorithm into that framework. For those consumers who are starting from scratch and face the task of building the application framework from ground zero, TI offers production-quality source code, called eXpressDSP Reference Frameworks (eRF), which uses both DSP/BIOS™ and XDAIS, to adapt a basic framework and quickly populate it with the desired XDAIS algorithm(s). The eRFs consist of production-worthy, reusable C-source code for both TI's C5000™ and C6000™ DSP families. Consumers can build on top of the blueprint code and relegate common elements such as memory management and channel encapsulation, allowing them to focus on specific system needs rather than the underlying foundation to achieve faster time-to-market. For those interested in using eRF, please refer to any of the technical notes from SPRA791 through SPRA799 (instead of this application note) as the starting point for leveraging the eXpressDSP™ Reference Frameworks.

The example used in this guide is a simple framework, written completely in the C language, and based on an audio application which applies a FIR filter algorithm to an input data stream of digitized audio samples. We will first analyze the existing framework and then outline each source code addition/modification to successfully integrate the XDAIS FIR algorithm. The sample algorithm used is the finite impulse response filter developed by TI's Technical Training Organization (TTO).

Contents

List of Figures

1 System View: Typical Structure of DSP Software Architecture

Here is a typical high-level overview of a DSP system which incorporates independent algorithm modules. The main application code typically sits on top of some core run-time support libraries, and in most cases, follows the C run-time conventions defined for the specific DSP architecture. The application, or system-specific code, communicates with each algorithm through the standard interfaces which all XDAIS algorithms must implement. The application also manages the rest of the system resources, such as I/O data streams and codecs. The application is in complete control and manages the entire system — algorithms are just like “slaves” driven by the “master” or framework of the application. Ideally, a DSP application should be broken down in this way (see Figure 1). XDAIS allows all compliant algorithms to ensure that multiple algorithms can interoperate and “play nicely” together.

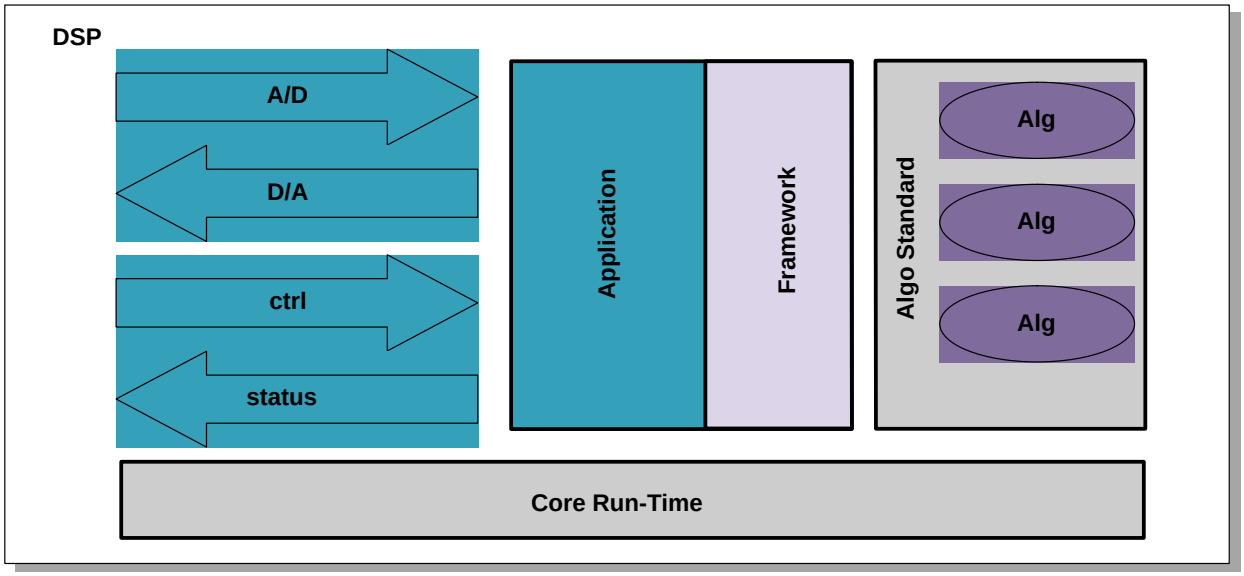

Figure 1. DSP Architecture and XDAIS-Compliant Algorithms

2 The FIR Audio Application Framework

The following source code reflects the FIR audio example which must be modified to integrate the XDAIS-compliant FIR_TTO algorithm:

```
void main (void)
{
      int i;

      initApplication();        /* Initialize DSP System */
      initInterrupts();         /* Initialize Interrupts */

      while(run)                /* 'Loop' – Wait Data */
      {
            if ( dataReadyFlag == READY )           /* Test Data Ready Flag    */
            {
                  dataReadyFlag = NOT_READY;        /* Reset Data Ready Flag*/

                  selectBuffer();                   /* Selects Ping or Pong Buffer */

                  if (filterEnable == 1)            /* Apply filter function */

                  {

                        /* copy input buffer into working buffer */
                        memcpy(workBuffer + (COEFF_SIZE – 1), processSrc,
                              FRAME_SIZE * sizeof(short));

                        /* call fir_gen to filter data */
                        fir_gen(workBuffer, filterCoeff[filterCutoff][0],
                               processDst, COEFF_SIZE, FRAME_SIZE);

            /* shift filter history to beginning of working buffer for next frame */
                        memcpy(workBuffer, workBuffer + FRAME_SIZE,
                              (COEFF_SIZE –1) * sizeof(short));

                  }

                  else                              /* Loop–Back if filter off */
                  {
                  for (i = 0; i < FRAME_SIZE ; i ++)
                      processDst[i] = processSrc[i];
                  }

                for( i = 0; i < FRAME_SIZE; i++ )
                        processDst[i] &= 0xfffe;    /* 0 LSB for codec */
            }

            /* Process other flags/interrupts */
      }
}
```

3 XDAIS Algorithm Deliverables

Every XDAIS-compliant algorithm will include the necessary header (interface) files, a single library archive containing the object files of the algorithm, and a single document which contains pertinent information to aid in the system integration effort. The vendor may also provide some 'helper' framework source files to include as part of the overall application to further expedite the integration process.

4 The Standard Interface (IALG)

Every XDAIS-compliant algorithm must implement a standard interface, called IALG, which is defined by the XDAIS specification (see the IALG_Fxns structure definition). This interface contains standard functions to allow the framework of an application to communicate with the algorithm module for memory management and algorithm instance creation, initialization, and deletion. The standard functions are shown here:

```
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;
```

The following is a summary of the **required** APIs and their purpose:

algAlloc() – called by the client to interrogate the algorithm for its memory requirements. The memory requirements are returned in a memory table which conveys the number of blocks of memory required, and the quality of each memory block (size, alignment, scratch/persistent, external/internal)

algInit() – called by the client to initialize the memories granted to each algorithm instance

algFree() – called by the client to interrogate an algorithm instance for its current memory holdings. The information is returned in a memory table in the same format used in the **algAlloc()** function. This allows the framework to dynamically reclaim the memory granted to an algorithm instance which is no longer needed.

In a **dynamic** memory system, where algorithm instance memory is allocated from a system heap, the **algAlloc(), algInit()**, and **algFree()** functions will all be used. In a **static** memory system, where algorithm instance memory is pre-defined at compile/link time, only the **algInit()** function needs to be called during runtime to initialize the memory before the algorithm can be run. In other words, **algAlloc()** and **algFree()** most likely will not be needed in a static memory scheme since these functions only aid in dynamic memory allocation/deallocation.

5 Framework Integration: Dynamic Memory Model

This section describes how to modify the non-XDAIS FIR audio framework to integrate the FIR_TTO XDAIS algorithm, for a dynamic memory model, where the memory is allocated and/or deallocated during run-time from/to one or more system memory heaps. Since memory management involves the standard API which is common across all compliant algorithms, we can leverage existing framework code geared specifically to interface with XDAIS algorithms. Hence, the individual coding effort will be minimal because framework source code (readily available from TI and Third Party vendors) can be added directly to the application.

In this example, we will leverage the TI-supplied framework code, algorithm reference framework (ALGRF). This code consists of higher-level concrete APIs, or ‘helper functions’ which take care of the memory management protocols between framework and algorithm to create, initialize, and delete algorithm instances during runtime of the application. Since every XDAIS-compliant algorithm must implement the standard IALG interface, the same framework code can be used with any standard algorithm.

The FIR_TTO algorithm package also comes with the highest-level MODULE framework code to further encapsulate and hide the details of the ALGRF code. This layer of code is referred to as the FIR interface. The combination of the FIR and ALGRF framework source code will allow for seamless, easy algorithm integration. The client code does not need to worry about all the different consumer and vendor-specific interfaces; by interfacing only with the FIR layer, modifications to the existing framework will be easy and clean. We assume the application is already DSP/BIOS-enabled to take advantage of being able to define multiple heaps on or off chip.

1. Include the header files that come with the XDAIS algorithm package, and the XDAIS system files.

```
#include <algrf.h>        /* TI–provided standard framework code */
#include “fir.h”          /* contains framework ‘helper’ functions */
#include “fir_tto.h”      /* “fir_tto.h” includes the “ifir.h” file */
```

2. Define a ‘params’ structure variable and an algorithm instance handle. Every compliant algorithm’s interface will have an I<MODULE>_Params structure which contains individual fields for the user to set to specify how the algorithm instance state should be created. For the FIR_TTO algorithm, the params structure definition looks like the following:

```
typedef struct IFIR_Params {
    Int size;/* must be first field of all params structures */
    Short *coeffPtr;
    Int filterLen;
    Int frameLen;
} IFIR_Params;
```

   Similarly, every compliant algorithm interface will define an algorithm object handle type (a handle is just a pointer to the algorithm instance); for the FIR_TTO algorithm it is

```
typedef struct IFIR_Obj *IFIR_Handle;
```

   Hence, the following 2 lines of code should be added at the top of the main() function:

```
FIR_Params firParams;
FIR_Handle firHandle;
```

3. Use the DSP/BIOS Configuration Tool to set up an internal and external memory heap for dynamic memory allocation during runtime. Call the algorithm ‘master module’ initialization function, as well as the init() and setup() functions from the ALGRF framework layer (only once in the beginning).

```
/* Memory segments (heaps) for XDAIS algorithms and MEM_alloc */
extern       int    IRAM;        /* internal data memory */
extern       int    SDRAM;       /* external data memory */

initApplication();               /* Initialize DSP System */
initInterrupts();                /* Initialize Interrupts */
FIR_TTO_init();                  /* Initialize TTO’s Filter as a whole */
ALGRF_init();                    /* Initialize Framework layer */
ALGRF_setup( IRAM, SDRAM, IRAM );/* Set up the memory heaps for ALGRF */
```

4. Set the algorithm’s instance creation parameters — these values will define the characteristics and ‘state’ of the instance. The structure will be passed directly to the algorithm instance before it is run.

```
/* Initialize algorithm creation parameters of Object */
firParams           = IFIR_PARAMS;
firParams.filterLen = COEFF_SIZE;
firParams.frameLen  = FRAME_SIZE;
firParams.coeffPtr  = (short *)&filterCoeff[filterCutoff][0];
```

5. Call the FIR_create() helper function, passing in the correct v-table symbol and the params structure.

```
firHandle = FIR_create(&FIR_TTO_IFIR, &firParams);
```

6. Call the FIR_apply() method to invoke the filter processing function.

```
FIR_apply(firHandle, processSrc, processDst);
```

7. To delete the algorithm instance, call the FIR_delete() function and pass in the handle as a parameter.

```
FIR_delete(firHandle);    /* Reclaim all memory blocks used by the instance */
```

8. During shutdown, call the algorithm 'master module' exit function for de-initialization purposes.

```
FIR_TTO_exit();           /* De-initialize Filter    */
```

The final code for a dynamic memory allocation scheme should look like the following:

```
#include <std.h>
#include <algrf.h>         /* TI-provided standard framework code */
#include "fir.h"           /* contains framework 'helper' functions */
#include "fir_tto.h"       /* "fir_tto.h" includes the "ifir.h" file */

/* Memory segments (heaps) for XDAIS algorithms and MEM_alloc */
extern       int    IRAM;        /* internal data memory */
extern       int    SDRAM;       /* external data memory */

void main (void)
{
       Int i;
       FIR_Params firParams;
       FIR_Handle firHandle;

       initApplication();              /* Initialize DSP System */
       initInterrupts();               /* Initialize Interrupts */
       FIR_TTO_init();                 /* Initialize TTO's Filter as a whole */
       ALGRF_init();                   /* Initialize Framework layer */
       ALGRF_setup( IRAM, SDRAM, IRAM );/* Set up the memory heaps for ALGRF */

       /* Initialize algorithm creation parameters of Object */
       firParams           = IFIR_PARAMS;
       firParams.filterLen = COEFF_SIZE;
       firParams.frameLen  = FRAME_SIZE;
       firParams.coeffPtr  = (short *)&filterCoeff[filterCutoff][0];

       firHandle = FIR_create(&FIR_TTO_IFIR, &firParams);

       if ( (firHandle != NULL) )
       {
             while(run)                              /* 'Loop' – Wait Data */
             {
                    if ( dataReadyFlag == READY )    /* Test Data Ready Flag */
                    {
                           dataReadyFlag = NOT_READY; /* Reset Data Ready Flag */

                           selectBuffer();           /* Selects Ping or Pong Buffer */

                           if (filterEnable == 1)    /* Apply filter function */

                                 FIR_apply(firHandle, processSrc, processDst);
                    }
                    else
                    {
                          for (i = 0; i < FRAME_SIZE ; i ++)
                                 processDst[i] = processSrc[i];
                    }
```

```
                    for( i = 0; i < FRAME_SIZE; i++ )
                           processDst[i] &= 0xfffe;  /* 0 the LSB for codec */

                    /* Process other flags/interrupts */
              }
        }

        FIR_delete(firHandle);    /* Reclaim all memory blocks used by the instance */

        FIR_TTO_exit();           /* De-initialize Filter   */
}
```

6 Framework Integration: Static Memory Model

This section describes how to modify the non-XDAIS FIR audio framework to integrate the FIR_TTO XDAIS algorithm, for a static memory model, where the memory is pre-allocated for each algorithm instance at compile/link time. The static framework requires less overhead than the dynamic framework, however, the framework code must be specifically tailored to the algorithm(s) to be integrated, since every algorithm will have unique memory requirements. In other words, creating the static framework requires more of a manual software coding process, which must be customized for each application. These steps are required:

1. Include the vendor-specific header file that comes with the XDAIS algorithm package.

   ```
   #include ”fir_tto.h”    /* ”fir_tto.h” includes the ”ifir.h” file */
   ```

2. Define a ‘params’ structure variable and an algorithm instance handle. Every compliant algorithm’s interface will have an I<MODULE>_Params structure which contains individual fields for the user to set to specify how the algorithm instance state should be created. For the FIR_TTO algorithm, the params structure definition looks like the following:

   ```
   typedef struct IFIR_Params {
       Int size; /* must be first field of all params structures */
       Short *coeffPtr;
       Int filterLen;
       Int frameLen;
   } IFIR_Params;
   ```

   Similarly, every compliant algorithm interface will define an algorithm object handle type (a handle is just a pointer to the algorithm instance); for the FIR_TTO algorithm it is

   ```
   typedef struct IFIR_Obj *IFIR_Handle;
   ```

   Hence, the following two lines of code should be added at the top of the main() function:

   ```
   IFIR_Params firParams;
   IFIR_Handle firHandle;
   ```

3. Call the algorithm’s ‘master module’ initialization function (only once in the beginning).

   ```
   initApplication();          /* Initialize DSP System */
   initInterrupts();           /* Initialize Interrupts */
   FIR_TTO_init();             /* Initialize TTO’s Filter as a whole */
   ```

4. Set the algorithm’s instance creation parameters — these values will define the characteristics and ‘state’ of the instance. The structure will be passed directly to the algorithm instance before it is run.

   ```
   /* Initialize algorithm creation parameters of Object */
   firParams           = IFIR_PARAMS;
   firParams.filterLen = COEFF_SIZE;
   firParams.frameLen  = FRAME_SIZE;
   firParams.coeffPtr  = (short *)&filterCoeff[filterCutoff][0];
   ```

5. Allocate the algorithm memory blocks and define a memory table based on information found in the documentation.

```
/* Buffers needed for the FIR filter's memTab that are define at link time */
/* These sizes can be determined from the specification provided with the  */
/* algorithm or by writing a simple program to determine the sizes         */
int    persistentBuffer0   [6];
short  persistentBuffer1   [COEFF_SIZE –1];
short  scratchBuffer       [FRAME_SIZE + COEFF_SIZE –1];

IALG_MemRec  memTab        [3]; /* Mem Tab structure declaration */
```

Where did we get this information? The Heap Data Memory table in the documentation shows how many blocks of memory are needed to create a single instance and the quality/characteristics of each memory block. The size of the memory table will just equal the number of memory blocks required. Now we need to set the 'base' fields of each memTab[] record to contain the location of the static memory addresses of each memory block. This is how the framework communicates, or grants the memory blocks to the algorithm instance:

```
/*  Initialize algorithm memory with static addresses of Object */
memTab[0 /* OBJECT     */ ].base  = persistentBuffer0;
memTab[1 /* HISTORYBUF */ ].base  = persistentBuffer1;
memTab[2 /* WORKBUF    */ ].base  = scratchBuffer;
```

6. Initialize the algorithm instance handle to point to the block of memory described by memTab[0].

```
firHandle = memTab[0].base;    /* Set fir handle to point to the instance object */
```

7. Initialize the pointer in the instance object to the global v-table symbol.

```
firHandle–>fxns = &FIR_TTO_IFIR;      /* Set pointer in instance object to vtab */
```

How do we know there is a 'fxns' field in the firHandle structure? In the "ifir.h" file, the object is defined. This is how the client code invokes the method of each algorithm instance, because every instance, by definition, will contain a pointer to the vector table of function pointers available to the user.

```
typedef struct IFIR_Obj {
    struct IFIR_Fxns *fxns;
} IFIR_Obj;
```

8. Call the algInit() function (through the v-table) to initialize all memories owned by the instance.

```
/*  Initialize Object */
firHandle–>fxns–>ialg.algInit( (IALG_Handle)firHandle,  memTab , NULL,
                               (IALG_Params *)&firParams );
```

The reason firHandle and firParams types must be typecast to IALG types is because the function prototype for the algInit() function uses IALG data types as defined in the Standard <ialg.h> system file:

```
typedef struct IALG_Obj *IALG_Handle;

typedef struct IALG_Params {
    Int size;       /* number of MAU in the structure */
} IALG_Params;

Int (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
```

9. Call the filter() function through the vector table (algorithm instance contains a pointer to it).

```
firHandle–>fxns–>filter(firHandle, processSrc, processDst);
```

The filter() function pointer was defined as part of the IFIR_Fxns v-table in the "ifir.h" file:

```
typedef struct IFIR_Fxns {
    IALG_Fxnsialg;    /* IFIR extends IALG */
```

```
    Void  (*filter)(IFIR_Handle handle, Short in[], Short out[]);
} IFIR_Fxns;
```

10. During shutdown, call the algorithm 'master module' exit function for de-initialization purposes.

```
FIR_TTO_exit();          /* De-initialize Filter     */
```

The final code for a static memory allocation scheme should look like the following:

```
#include <std.h>
#include "fir_tto.h"         /* "fir_tto.h" includes the "ifir.h" file */

/* Buffers need for the fir filter's memTab that are define at link time */
/* These sizes can be determined from the specification provided with the */
/* algorithm or by writing a simple program to determine the sizes        */
int    persistentBuffer0   [6];
short  persistentBuffer1   [COEFF_SIZE -1];
short  scratchBuffer       [FRAME_SIZE + COEFF_SIZE -1];

IALG_MemRec  memTab        [3];   /* memTab structure declaration    */


void main (void)
{
       Int i;
       IFIR_Params firParams;
       IFIR_Handle firHandle;

       initApplication();         /* Initialize DSP System */
       initInterrupts();          /* Initialize Interrupts */
       FIR_TTO_init();            /* Initialize TTO's Filter as a whole */

       /* Initialize algorithm creation parameters of Object */
       firParams           = IFIR_PARAMS;
       firParams.filterLen = COEFF_SIZE;
       firParams.frameLen  = FRAME_SIZE;
       firParams.coeffPtr  = (short *)&filterCoeff[filterCutoff][0];

       /*  Initialize algorithm memory with static addresses of Object */
       memTab[0 /* OBJECT     */ ].base  = persistentBuffer0;
       memTab[1 /* HISTORYBUF */ ].base  = persistentBuffer1;
       memTab[2 /* WORKBUF    */ ].base  = scratchBuffer;

       firHandle = memTab[0].base;/* Set fir handle to point to the instance object */

       firHandle->fxns = &FIR_TTO_IFIR; /* Set pointer in instance object to vtab */

       /*  Initialize Object */
       firHandle->fxns->ialg.algInit( (IALG_Handle)firHandle,  memTab , NULL,
                                      (IALG_Params *)&firParams );

       if ( (firHandle != NULL) )
       {
             while(run)                              /* 'Loop' - Wait Data */
             {
                    if ( dataReadyFlag == READY )    /* Test Data Ready Flag */
                    {
                          dataReadyFlag = NOT_READY; /* Reset Data Ready Flag */

                          selectBuffer();            /* Selects Ping or Pong Buffer */
```

```
                        if (filterEnable == 1)     /* Apply filter function */

                          firHandle->fxns->filter(firHandle, processSrc, processDst);
                        }
                        else
                        {
                          for (i = 0; i < FRAME_SIZE ; i ++)
                                processDst[i] = processSrc[i];
                        }

                        for( i = 0; i < FRAME_SIZE; i++ )
                              processDst[i] &= 0xfffe;  /* 0 the LSB for codec */

                        /* Process other flags/interrupts */
            }
        }

        FIR_TTO_exit();           /* De-initialize Filter */
}
```

To make the code more readable and easier to maintain, an ALGRF_new() function could be created to encapsulate the lines of code needed to create any static memory instance.

```
/*
 *  Instantiate a 'static' XDAIS algorithm.
 *  Very low overhead method of bringing up an algorithm. No memory heap
 *  usage, no buffer allocations. Initializes algorithm to use buffers
 *  whose alignment, size etc have been preconfigured to match algorithm needs
 */

IALG_Handle ALGRF_new(IALG_Fxns *fxns, IALG_Params *params,
        Char *algChanBufs[], SmUns numAlgChanBufs)
{
    /* Use a large enough memTab structure for any algorithm */
    IALG_MemRec memTab[ALGMIN_MAXNUMRECS];
    IALG_Handle alg;
    Int i;

    /*  Initialize algo memory with preconfigured static buffer addresses */
    for (i=0; i<numAlgChanBufs; i++)    /* loop thro num memTab's of algo */
    {
        memTab[i].base = algChanBufs[i];
    }

    /* bind handle to algorithm's instance object and function table */
    alg = memTab[0].base;
    alg->fxns = fxns;

    if (fxns->algInit(alg, memTab, NULL, params) == IALG_EOK) {
        return (alg);   /* algInit SUCCESS: return handle to algo */
    }
    else {
        return (NULL);  /* algInit FAILURE: return NULL */
    }
}
```

Then, a higher-level FIR_new() function could be created to hide the typecasts required to call the ALGRF_new() function:

```
/*
 *  ======== FIR_new ========
 *  Statically create a FIR instance ie no heaps etc
 */
FIR_Handle FIR_new(const IFIR_Fxns *fxns,
        FIR_Params *params, Char *firChanBufs[], SmUns firNumChanBufs)
{
    return ( (FIR_Handle)ALGRF_new((IALG_Fxns *)fxns,
        (IALG_Params *)params, firChanBufs, firNumChanBufs) );
}
```

More readable and maintainable code for a static memory allocation scheme now looks like the following:

```
#include <std.h>
#include ”fir_tto.h”       /* ”fir_tto.h” includes the ”ifir.h” file */

/* Buffers need for the fir filter’s memTab that are define at link time  */
/* These sizes can be determined from the specification provided with the */
/* algorithm or by writing a simple program to determine the sizes        */
int    persistentBuffer0   [6];
short  persistentBuffer1   [COEFF_SIZE –1];
short  scratchBuffer       [FRAME_SIZE + COEFF_SIZE –1];

IALG_MemRec  memTab        [3];   /* memTab structure declaration    */

void main (void)
{
       Int i;
       IFIR_Params firParams;
       FIR_Handle firHandle;
       Char *firChanBufs[] = { persistentBuffer0, persistentBuffer1, scratchBuffer };

       initApplication();           /* Initialize DSP System */
       initInterrupts();            /* Initialize Interrupts */
       FIR_TTO_init();              /* Initialize TTO’s Filter as a whole */

       /* Initialize algorithm creation parameters of Object */
       firParams           = FIR_PARAMS;
       firParams.filterLen = COEFF_SIZE;
       firParams.frameLen  = FRAME_SIZE;
       firParams.coeffPtr  = (short *)&filterCoeff[filterCutoff][0];

       /* Statically create a FIR instance ie no heaps etc */
       firHandle = FIR_new(&FIR_TTO_IFIR, firParams,
                           firChanBufs, (sizeof(firChanBufs) / sizeof(firChanBufs[0])));

       if ( (firHandle != NULL) )
       {
             while(run)                                /* ’Loop’ – Wait Data */
             {
                    if ( dataReadyFlag == READY )     /* Test Data Ready Flag */
                    {
                           dataReadyFlag = NOT_READY; /* Reset Data Ready Flag */

                           selectBuffer();            /* Selects Ping or Pong Buffer */

                           if (filterEnable == 1)     /* Apply filter function */

                             firHandle–>fxns–>filter(firHandle, processSrc, processDst);
                    }
                    else
                    {
                           for (i = 0; i < FRAME_SIZE ; i ++)
                                 processDst[i] = processSrc[i];
                    }
```

```
                    for( i = 0; i < FRAME_SIZE; i++ )
                          processDst[i] &= 0xfffe;    /* 0 the LSB for codec */

                    /* Process other flags/interrupts */
                    }
        }

        FIR_TTO_exit();    /* De-initialize Filter    */
}
```

In addition, TI provides a special tool called XDAIS Optimizer for Static Applications, (DOSA), a Code Composer Studio plug-in, that is used to automate the steps involved in optimizing a XDAIS algorithm for use in static memory systems. The tool automatically removes all the overhead that is not required for a static memory system, and performs design-time object creation/initialization of the algorithm modules. If you would like to further explore the possibility of reducing the algorithm overhead to zero, please refer to the application note, *Achieving Zero Overhead with the TMS320™ DSP Algorithm Standard IALG Interface* (SPRA716).

7 Conclusion

On the surface, XDAIS algorithm integration may seem complicated, but the benefits of integrating an algorithm module which is guaranteed to interoperate with any other software module in the system is clearly a valuable benefit. And because all XDAIS-compliant algorithms must implement the standard IALG interface for client-algorithm memory management, the same framework code can be used across all standardized algorithms. Additionally, the same set of code can be used in both static and dynamic memory systems, providing the XDAIS consumer with the benefit of complete flexibility of how algorithms are incorporated into the final DSP system application.

Appendix A FIR_TTO Documentation

This section contains a set of tables that help characterize the FIR_TTO algorithm. This information can be extremely useful to the system integrator who is trying to integrate algorithms into a system that has limited memory.

Module	Vendor	Variant	Arch	Model	Version	Doc Date	Library Name
FIR	TTO	None	62	Little Endian		04.15.2000	fir_tto.l62

ROMable (Rule 5)

Yes	No
X	

Heap Data Memory (Rule 19)

memTab	Attribute	Size (bytes)	Align (MAUs)	Space
0	Persist	24	4	External
1	Scratch	2 * (filterLen – 1)	2	External
2	Scratch	2 * [(filterLen – 1) + frameLen]	2	DARAM0

Note: The unit for size is (8–bit) byte and the unit for align is Minimum Addressable Unit (MAUs).

Stack Space Memory (Rule 20)

	Size (bytes)	Align (MAUs)
Worst Case	64	0

Static Data Memory (Rule 21)

.data Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch	.far Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch
fir_tto_vt.obj	48	0	R	No	fir_tto_vt.obj	40	0	R	No

Program Memory (Rule 22)

	Code	
Code Sections	Size (bytes)	Align (MAUs)
`.text:algAlloc`	288	0
`.text:algInit`	192	0
`.text:algFree`	192	0
`.text:algActivate`	64	0
`.text:algMoved`	96	0

Program Memory (Rule 22) (Continued)

Code Sections	Code Size (bytes)	Code Align (MAUs)
.text:algDeactivate	64	0
.text:algControl	128	0
.text:algNumAlloc	32	0
.text:init	32	0
.text:exit	32	0
.text:filter	512	0

Interrupt Latency (Rule 23)

Operation	Typical Call Frequency (microsec)	Worst-case (Instruction Cycles)
filter()	On Demand	0

Operation	Typical Call Frequency (microsec)	Worst-case Cycles/Period
filter()	On Demand	(4 * frameLen) + (filterLen * frameLen) + 1025

Additional notes regarding TMS320 DSP algorithm standard compliance:

- Rule 1: This algorithm follows the run-time conventions imposed by TI's implementation of the C programming language.
- Rule 2: This algorithm is re-entrant within a preemptive environment (including time-sliced preemption).
- Rule 3: All algorithm data references are fully relocatable.
- Rule 4: All algorithm code is fully relocatable.
- Rule 6: This algorithm does not directly access any peripheral device.
- Rule 10: This algorithm follows the naming conventions of the DSP/BIOS for all external declarations.
- Rule 25: This algorithm was compiled in little endian mode (C6x only).
- Rule 26: This algorithm accesses all static and global data as far data (C6x only).
- Rule 27: This algorithm operates properly with program memory operated in cache mode (C6x only).

Appendix B Algorithm Header Files

This section contains all the header files that are included with the FIR_TTO XDAIS Algorithm Package.

```
/*
 *  ======== ifir.h ========
 *  IFIR Interface Header
 */
#ifndef IFIR_
#define IFIR_

#include <ialg.h>

/*
 *  ======== IFIR_Handle ========
 *  This handle is used to reference all FIR instance objects
 */
typedef struct IFIR_Obj *IFIR_Handle;

/*
 *  ======== IFIR_Obj ========
 *  This structure must be the first field of all FIR instance objects
 */
typedef struct IFIR_Obj {
    struct IFIR_Fxns *fxns;
} IFIR_Obj;

/*
 *  ======== IFIR_Status ========
 *  Status structure defines the parameters that can be changed or read
 *  during real-time operation of the alogrithm.
 */
typedef struct IFIR_Status {
    Int size;     /* must be first field of all status structures */
    Short *coeffPtr; /* READ/WRITE */

} IFIR_Status;
/*
 *  ======== IFIR_Cmd ========
 *  The Cmd enumeration defines the control commands for the FIR
 *  control method.
 */
typedef enum IFIR_Cmd {
       IFIR_GETSTATUS,
       IFIR_SETSTATUS
} IFIR_Cmd;

/*
 *  ======== IFIR_Params ========
 *  This structure defines the creation parameters for all FIR objects
 */
typedef struct IFIR_Params {
    Int size;     /* must be first field of all params structures */
    Short *coeffPtr;
    Int filterLen;
    Int frameLen;
} IFIR_Params;

/*
 *  ======== IFIR_PARAMS ========
 *  Default parameter values for FIR instance objects
 */
extern IFIR_Params IFIR_PARAMS;

/*
 *  ======== IFIR_Fxns ========
 *  This structure defines all of the operations on FIR objects
 */
```

```
typedef struct IFIR_Fxns {
    IALG_Fxns ialg;    /* IFIR extends IALG */
    Void  (*filter)(IFIR_Handle handle, Short in[], Short out[]);

} IFIR_Fxns;

#endif /* IFIR_ */

/*
 *  ======== fir_tto.h ========
 *  Interface for the FIR_TTO module; TTO's implementation of the IFIR interface
 */
#ifndef FIR_TTO_
#define FIR_TTO_

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

/*
 *  ======== FIR_TTO_IALG ========
 *  TTO's implementation of the IALG interface for FIR
 */
extern IALG_Fxns FIR_TTO_IALG;

/*
 *  ======== FIR_TTO_IFIR ========
 *  TTO's implementation of the IFIR interface
 */
extern IFIR_Fxns FIR_TTO_IFIR;

#endif /* FIR_TTO_ */
```

Appendix C ALGRF Header and Source Code

This section contains all the header and source files for the ALGRF standard framework provided by TI. The entire ALGRF source code can be obtained by downloading the eXpressDSP Reference Frameworks from www.dspvillage.com.

```
/*
 *  ======== algrf.h ========
 */
#ifndef ALGRF_
#define ALGRF_

#include <std.h>

#include <ialg.h>

#ifdef __cplusplus
extern ”C” {
#endif

typedef IALG_Handle ALGRF_Handle;

typedef struct ALGRF_Config {
    Int INTHEAP;
    Int EXTHEAP;
    Int MEMTABHEAP;
} ALGRF_Config;

extern ALGRF_Config* ALGRF;  /* declared in instalg_setup.c */

/*
 *  ======== ALGRF_memSpace ========
 */
static inline Int ALGRF_memSpace(IALG_MemSpace space)
{
    switch (space) {
       case IALG_DARAM0:
       case IALG_DARAM1:
       case IALG_SARAM:    /* IALG_SARAM0 same as IALG_SARAM */
       case IALG_SARAM1:
       case IALG_DARAM2:
       case IALG_SARAM2: {
           return (ALGRF->INTHEAP);
       }

       case IALG_ESDATA:
       case IALG_EXTERNAL: {
           return (ALGRF->EXTHEAP);
       }

       default: {
           return (ALGRF->EXTHEAP);
       }
    }
}

/* Initialize scratch memory buffers before processing */
extern Void ALGRF_activate(ALGRF_Handle alg);

/* Algorithm specific control and status */
extern Int ALGRF_control(ALGRF_Handle alg, IALG_Cmd cmd,
    IALG_Status *statusPtr);

/*
 *  Algorithm instance creation API for applications not supporting
 *  shared scratch memory buffers
 */
extern ALGRF_Handle ALGRF_create(IALG_Fxns *fxns, IALG_Handle parent,
    IALG_Params *params);
```

```
/*
 *  Algorithm instance creation API for applications supporting shared
 *  scratch memory buffers
 */
extern ALGRF_Handle ALGRF_createScratchSupport(IALG_Fxns *fxns,
    IALG_Handle parent, IALG_Params *params, Void *scratchBuf,
    Int scratchSize);

/* Save all persistent data to non–scratch memory */
extern Void ALGRF_deactivate(ALGRF_Handle alg);

/*
 *  Algorithm instance deletion API for applications not supporting
 *  shared scratch memory buffers
 */
extern Bool ALGRF_delete(ALGRF_Handle instalg);

/*
 *  Algorithm instance deletion API for applications supporting
 *  shared scratch memory buffers
 */
extern Bool ALGRF_deleteScratchSupport(ALGRF_Handle instalg);

/* Free all memory allocated to an algorithm */
extern Void ALGRF_memFree(IALG_MemRec memTab[], Int numRecs);

/* Free all memory allocated to an algorithm, except internal scratch */
extern Void ALGRF_memFreeScratchSupport(IALG_MemRec memTab[], Int numRecs);

/* Configure the system to use the selected heaps */
extern Void ALGRF_setup(Int internalHeap, Int externalHeap,
    Int memTabHeap);

#ifdef __cplusplus
}
#endif /* extern ”C” */

#endif  /* ALGRF_ */


/*
 *  ======== algrf_cre.c ========
 */

#pragma CODE_SECTION(ALGRF_create, ”.text:create”)
#pragma CODE_SECTION(ALGRF_memFree, ”.text:create” )
#pragma CODE_SECTION(memAlloc, ”.text:create”)

#include <std.h>

#include <ialg.h>
#include <algrf.h>
#include <mem.h>


static Bool memAlloc(IALG_MemRec memTab[], Int numRecs);

/*
 *  ======== ALGRF_new ========
 *
 *  Instantiate a ’static’ XDAIS algorithm.
 *  Very low overhead method of bringing up an algorithm. No memory heap
 *  usage, no buffer allocations. Initializes algorithm to use buffers
 *  whose alignment, size etc have been preconfigured to match algorithm needs
 *
 */
IALG_Handle ALGRF_new(IALG_Fxns *fxns, IALG_Params *params,
        Char *algChanBufs[], SmUns numAlgChanBufs)
{
    /* Use a large enough memTab structure for any algorithm */
```

```
    IALG_MemRec memTab[ALGMIN_MAXNUMRECS];
    IALG_Handle alg;
    Int i;

    /*  Initialize algo memory with preconfigured static buffer addresses */
    for (i=0; i<numAlgChanBufs; i++)    /* loop thro num memTab's of algo */
    {
        memTab[i].base = algChanBufs[i];
    }

    /* bind handle to algorithm's instance object and function table */
    alg = memTab[0].base;
    alg->fxns = fxns;

    if (fxns->algInit(alg, memTab, NULL, params) == IALG_EOK) {
        return (alg);   /* algInit SUCCESS: return handle to algo */
    }
    else {
        return (NULL);  /* algInit FAILURE: return NULL */
    }
}

/*
 *  ======== ALGRF_create ========
 *  Create an instance of an algorithm
 */
ALGRF_Handle ALGRF_create(IALG_Fxns *fxns, IALG_Handle parent,
    IALG_Params *params)
{
    IALG_MemRec *memTab;
    Int numRecs, sizeMemTab;
    ALGRF_Handle alg;
    IALG_Fxns *fxnsPtr;

    if (fxns != NULL) {
        numRecs = (fxns->algNumAlloc != NULL) ?
            fxns->algNumAlloc() : IALG_DEFMEMRECS;

        sizeMemTab = numRecs * sizeof(IALG_MemRec);

        /* allocate memory for the memTab structure */
        memTab = (IALG_MemRec *)MEM_alloc(ALGRF->MEMTABHEAP, sizeMemTab, 0);

        if (memTab != MEM_ILLEGAL) {
            /* query the algorithm about its memory requirements */
            numRecs = fxns->algAlloc(params, &fxnsPtr, memTab);

            if (numRecs <= 0 ) {
                return (NULL);
            }

            if ((memAlloc(memTab, numRecs)) == TRUE) {
                alg = (IALG_Handle)memTab[0].base;
                alg->fxns = fxns;

                /*
                 *  now that memAlloc has allocated memory for each memory
                 *  block that the algorithm requested, call algInit so the
                 *  algorithm can initialize the memory
                 */
                if (fxns->algInit(alg, memTab, parent, params) == IALG_EOK) {
                    /*
                     *  if algInit successful, free the memTab structure
                     *  and return a handle to the algorithm
                     */
                    MEM_free(ALGRF->MEMTABHEAP, memTab, sizeMemTab);
                    return (alg);
                }

                /*
```

```
                 *  algInit unsuccessful, free the memory allocated for
                 *  the algorithm and the memTab structure
                 */
                fxns->algFree(alg, memTab);
                ALGRF_memFree(memTab, numRecs);
                MEM_free(ALGRF->MEMTABHEAP, memTab, sizeMemTab);
            }
        }
    }

    return (NULL);
}

/*
 *  ======== ALGRF_memFree ========
 *  Free the memory allocated for an algorithm
 *
 */
Void ALGRF_memFree(IALG_MemRec memTab[], Int numRecs)
{
    Int i;

    for (i = 0; i < numRecs; i++) {
        if (memTab[i].base != MEM_ILLEGAL) {
            MEM_free(ALGRF_memSpace(memTab[i].space), memTab[i].base,
                memTab[i].size);
        }
    }
}

/*
 *  ======== memAlloc ========
 *  Allocate memory for an algorithm
 */
static Bool memAlloc(IALG_MemRec memTab[], Int numRecs)
{
    Int i;
    /*
     *  algorithm standard specifies that memTab[0] must be
     *  initialized with zeros
     */
    if ((memTab[0].base = MEM_calloc(ALGRF_memSpace(memTab[0].space),
        memTab[0].size, memTab[0].alignment)) == MEM_ILLEGAL) {
        return (FALSE);
    }

    for (i = 1; i < numRecs; i++) {

        memTab[i].base = MEM_alloc(ALGRF_memSpace(memTab[i].space),
            memTab[i].size, memTab[i].alignment);

        if (memTab[i].base == MEM_ILLEGAL) {
            /*
             *  if any memory allocation fails, free all memory previously
             *  allocated for the algorithm
             */
            ALGRF_memFree(memTab, i);
            return (FALSE);
        }
    }

    return (TRUE);
}

/*
 *  ======== algrf_setup.c ========
 */

#pragma CODE_SECTION(ALGRF_setup, ".text:setup")
```

```
#include <std.h>

#include <algrf.h>

ALGRF_Config ALGRF_config = {
      0,  /* default segment for Internal Heap */
      0,  /* default segment for External Heap */
      0   /* default segment for memTab structure allocation */
};

ALGRF_Config* ALGRF = &ALGRF_config;

/*
 *  ======== ALGRF_setup ========
 *  Configure the system to use the selected heaps
 */
Void ALGRF_setup(Int internalHeap, Int externalHeap, Int memTabHeap)
{
    ALGRF->INTHEAP = internalHeap;
    ALGRF->EXTHEAP = externalHeap;
    ALGRF->MEMTABHEAP = memTabHeap;
}

/*
 *  ======== algrf_del.c ========
 */

#pragma CODE_SECTION(ALGRF_delete, ".text:delete")

#include <std.h>

#include <ialg.h>
#include <algrf.h>

#include <mem.h>

/*
 *  ======== ALGRF_delete ========
 *  Delete an instance of an algorithm
 */
Bool ALGRF_delete(ALGRF_Handle alg)
{
    IALG_MemRec *memTab;
    Int numRecs, sizeMemTab;
    IALG_Fxns *fxns;

    if (alg == NULL || alg->fxns == NULL) {
        return (FALSE);
    }

    fxns = alg->fxns;
    numRecs = (fxns->algNumAlloc != NULL) ?
        fxns->algNumAlloc() : IALG_DEFMEMRECS;

    sizeMemTab = numRecs * sizeof(IALG_MemRec);

    /* allocate memory for the memTab structure */
    if ((memTab = (IALG_MemRec *)MEM_alloc(ALGRF->MEMTABHEAP, sizeMemTab, 0))
        == MEM_ILLEGAL) {
        return (FALSE);
    }

    memTab[0].base = alg;

    /*
     *  query the algorithm about its memory allocations, and
     *  then free the memory allocated to the algorithm
     */
    numRecs = fxns->algFree(alg, memTab);
    ALGRF_memFree(memTab, numRecs);
```

```
    /* Free the memTab structure */
    MEM_free(ALGRF->MEMTABHEAP, memTab, sizeMemTab);

    return (TRUE);
}

/*
 *  ======== ALG_init ========
 */
Void ALGRF_init(Void)
{
}

/*
 *  ======== ALGRF_exit ========
 */
Void ALGRF_exit(Void)
{
}
```

Appendix D “FIR CONCRETE API” Header & Source Code

These files are not functions are part of the actual algorithm library, but are provided at the option of the vendor to aid the consumer. These files are provided for frameworks which are modeled after a dynamic memory allocation scheme. They can be added directly into the application project to further expedite the integration process.

```
/*
 *  ======== fir.h ========
 *  This header defines the interface used by clients of the FIR module
 */
#ifndef FIR_
#define FIR_

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


/*
 *  ======== FIR_Handle ========
 *  This pointer is used to reference all FIR instance objects
 */
typedef struct IFIR_Obj *FIR_Handle;

/*
 *  ======== FIR_Params ========
 *  This structure defines the creation parameters for all FIR objects
 */
typedef IFIR_Params FIR_Params;

/*
 *  ======== FIR_PARAMS ========
 *  This structure defines the default creation parameters for FIR objects
 */
#define FIR_PARAMS   IFIR_PARAMS

/*
 *  ======== FIR_Status ========
 *  This structure defines the real-time parameters for FIR objects
 */
typedef struct IFIR_Status   FIR_Status;

/*
 *  ======== FIR_Cmd ========
 *  This typedef defines the control commands FIR objects
 */
typedef IFIR_Cmd   FIR_Cmd;

/*
 * ===== control method commands =====
 */
#define FIR_GETSTATUS IFIR_GETSTATUS
#define FIR_SETSTATUS IFIR_SETSTATUS

/*
 *  ======== FIR_create ========
 *  Create an FIR instance object (using parameters specified by prms)
 */
extern FIR_Handle FIR_create(const IFIR_Fxns *fxns, const FIR_Params *prms);

/*
 *  ======== FIR_new ========
 *  Initialize an FIR instance object with static memory (using parameters in prms)
 */

extern FIR_Handle FIR_new(const IFIR_Fxns *fxns,
        FIR_Params *params, Char *firChanBufs[], SmUns firNumChanBufs)
```

```
/*
 *  ======== FIR_delete ========
 *  Delete the FIR instance object specified by handle
 */
extern Void FIR_delete(FIR_Handle handle);

/*
 *  ======== FIR_filter ========
 */
extern Void  FIR_filter(FIR_Handle handle, Short in[], Short out[]);

/*
 *  ======== FIR_apply ========
 */
extern Void  FIR_apply(FIR_Handle handle, Short in[], Short out[]);

#endif /* FIR_ */


/*
 *  ======== fir.c ========
 *  This file implements all methods defined in fir.h
 */
#pragma CODE_SECTION(FIR_create,  ".text:create")
#pragma CODE_SECTION(FIR_delete,  ".text:delete")
#pragma CODE_SECTION(FIR_init,  ".text:init")
#pragma CODE_SECTION(FIR_exit,  ".text:exit")
#pragma CODE_SECTION(FIR_filter,  ".text:filter")
#pragma CODE_SECTION(FIR_apply,  ".text:apply")
#pragma CODE_SECTION(FIR_control,  ".text:control")

#include <std.h>
#include <algrf.h>
#include <ialg.h>
#include "ifir.h"
#include "fir.h"

#define FIR_init_ FIR_TTO_init
#define FIR_exit_ FIR_TTO_exit

/*
 *  ======== FIR_create ========
 *  Create an FIR instance object (using parameters specified by prms)
 */
FIR_Handle FIR_create(const IFIR_Fxns *fxns, const FIR_Params *prms)
{
    return ((FIR_Handle)ALGRF_create((IALG_Fxns *)fxns, NULL, (IALG_Params *)prms));
}

/*
 *  ======== FIR_new ========
 *  Statically create a FIR instance ie no heaps etc
 */
FIR_Handle FIR_new(const IFIR_Fxns *fxns,
        FIR_Params *params, Char *firChanBufs[], SmUns firNumChanBufs)
{
    return ( (FIR_Handle)ALGRF_new((IALG_Fxns *)fxns,
        (IALG_Params *)params, firChanBufs, firNumChanBufs) );
}


/*
 *  ======== FIR_delete ========
 *  Delete the FIR instance object specified by handle
 */
Void FIR_delete(FIR_Handle handle)
{
    ALGRF_delete((ALGRF_Handle)handle);
}

/*
```

```
 *  ======== FIR_init ========
 *  FIR module initialization
 */
Void  FIR_init(Void)
{
       FIR_init_();
}

/*
 *  ======== FIR_exit ========
 *  FIR module finalization
 */
Void  FIR_exit(Void)
{
       FIR_exit_();
}

/*
 *  ======== FIR_filter ========
 */
Void  FIR_filter(FIR_Handle handle, Short in[], Short out[])
{

    handle->fxns->filter(handle, in, out);

}

/*
 *  ======== FIR_apply ========
 */
Void  FIR_apply(FIR_Handle handle, Short in[], Short out[])
{
    ALGRF_activate((ALGRF_Handle)handle);

    handle->fxns->filter(handle, in, out);

    ALGRF_deactivate((ALGRF_Handle)handle);
}

/*
 *  ======== FIR_control ========
 */
Int FIR_control(FIR_Handle handle, FIR_Cmd cmd, FIR_Status *statusPtr)
{
    Int i;

    ALGRF_activate((ALGRF_Handle)handle);

    i = handle->fxns->ialg.algControl((IALG_Handle)handle, (IALG_Cmd)cmd,
                                      (IALG_Status*)statusPtr);

    ALGRF_deactivate((ALGRF_Handle)handle);

    return i;
}
```

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265