Introduction

In this module, you will learn how to incorporate an XDAIS-compliant algorithm into your application.

Outline

- Code Integration Problems
- Background Terminology
- Basic XDAIS Components
- XDAIS Example – Sine Wave Algorithm
- Algorithm Instance Lifecycle
- Lab 11 – Using a XDAIS FIR Algorithm
- Additional Topics

Goals for Lab 11

- Add a xDAIS FIR Filter to system
- Filter used to eliminate sinewave from audio stream
Chapter Topics

Using a XDAIS Algorithm .......................................................................................................................11-1

Code Integration Problems .....................................................................................................................11-3
 Three Integration Issues ........................................................................................................................11-3
 Traditional Solutions ..........................................................................................................................11-4
 TI XDAIS Solution ...................................................................................................................................11-5

Background Terminology .......................................................................................................................11-6

Basic XDAIS Components .......................................................................................................................11-9

XDAIS Example – Sinewave Algorithm ...................................................................................................11-11

Algorithm Instance Lifecycle ......................................................................................................................11-13

Lab 11 – Integrating an XDAIS algorithm ...............................................................................................11-19

Lab 11 Procedure ....................................................................................................................................11-20
 Examine and Edit xdais.c .......................................................................................................................11-20
 Apply the FIR Filter to the Audio .........................................................................................................11-23
 Add Files to Project ................................................................................................................................11-24
 Build and the Run program ..................................................................................................................11-25

Additional Topics ..................................................................................................................................11-27
 XDAIS Rules and Guidelines ..................................................................................................................11-27
 XDAIS Certification .................................................................................................................................11-28
 XDAIS Third Party Support ....................................................................................................................11-29
 Creating a XDAIS Algorithm with Component Wizard ........................................................................11-29
Code Integration Problems

Three Integration Issues

1. Using Multiple Algorithms

What problems might occur when integrating two different algorithms into an application?

- Will one use the memory required by another?
  
  For example:
  
  Given limited fast internal memory, will one fail if another takes too much?

- What if one algorithm uses an interrupt or EDMA channel required by another?

- Basically, any system resource can cause integration problems between algorithms.

2. Using the Same Algorithm Multiple Times

sine1.c

```c
float FreqTone, FreqSampleRate;
static float A, y[3];
void SINE_init()
short SINE_Value()
void SINE_blockFill()
```

- What would happen if you reused the sine algorithm for a 2nd sine wave tone?
  
  - Variable names **conflict**
  
  - May need to rewrite functions to handle two (or more) tones

sine2.c

```c
float FreqTone, FreqSampleRate;
static float A, y[3];
void SINE_init()
short SINE_Value()
void SINE_blockFill()
```

Note:

- This very problem occurred to the sine algorithm when we introduced multi-channel (sorting) in CH 7.

- After this chapter, we could drastically improve our solution.

And finally ...
3. Buying Algorithms

Why is it hard to integrate someone else’s algo?

1. Will the function names conflict with other code in the system?
2. Will it use memory or peripherals needed by other algo’s?
3. How can I run the same algo on more than one channel at a time? (How can I prevent variables from conflicting?)
4. Don't know how fast it runs …
   … or how much memory it uses.
5. How can I adapt the algorithm to meet my needs?
6. How many interfaces (API's) do I have to learn?

We’ve already seen the first three, four thru six are specific to using someone else’s code …

What's the solution?

Traditional Solutions

Traditional Solutions

1. Manually integrate algorithms together by finding all (hopefully) the conflicts and fixing them.

2. Rather than reusing an algorithm (e.g. our sinewave), rewrite algorithm to provide the number of required channels.

3. When I buy an algorithm, “I need the source code or I can’t guarantee my application will work.”
   - Without source code (and lots of development time), I can’t use the first two methods of code integration.
   - But, purchasing source code costs a lot of money!

What's the alternative?
TI XDAIS Solution

- Modularize algorithms. That is, use a standard interface between: Application ↔ Algorithms
- TI designed a DSP algorithm interface: XDAIS
  - Public, published set of rules & guidelines
  - Algorithm certification
  - Only one interface to learn for all algorithms and vendors!

TI DSP Algorithm Standard (XDAIS)

- Off-the-shelf DSP content
- Ease of integration
- Purchase once, use widely

Application Developers

- Rules & Guidelines Applied to Algorithm Software Modules
  - Programming Rules
  - Standard Interface Defined by TI
  - Algorithm Packaging
  - Algorithm Performance

TEXAS INSTRUMENTS

TMS320™ DSP Algorithm Standard Specification (XDAIS)

APPLICATION DEVELOPERS

ALGORITHM PRODUCERS

- Write once, deploy widely
- Or, sell widely

XDAIS is the standard interface specification for DSP algorithms
**What is an Instance?**

- This is a key concept in XDAIS
- To demonstrate the concept, let’s examine an “instance” in C code

```c
typedef struct myType {
    int var1;
    short var2;
    char var3;
};
```

```c
myType myVar;
myType anotherVar;
```

**Define Datatype**
- Only a “template”
- No memory allocated

**Create an Instance of that Datatype**
- Memory is allocated
- Can create multiple instances

---

**What is an Interface?**

- “Interface” can mean many things
- We define it conceptually for the purposes of this chapter
- Let’s start by defining a Function interface

```c
int myFunction(short a, int b){
    return((int)a + b);
}
```

**Example Function**

**Function Interface:**
- Describes how the function is used
- That is, how does an application interface to it

Extending this definition,
How does an algo differ from a function?
**What is an Algorithm?**

- Algo’s usually are more than just a single function, an algorithm may include:
  - Data Types
  - Data Objects
  - Functions

```c
typedef struct myType {}
myType var1;
int var2;
int myFunction(short a, int b)
```

- An Algorithm’s Interface then must include a description of all the:
  - functions,
  - data types, and
  - data objects

available to the application using the algorithm

- Often, this is called an API or Application Programming Interface

---

- We could think of wrapping all these parts of an algorithm into a code module
- Or better yet, let’s just use the term module
- In other words, we use the term module when speaking abstractly about any algorithm

How can we describe an algorithm’s interface?
Background Terminology

Module (Algorithm) Interface

- What is an Algorithm’s Interface (i.e. Module Interface)?
  - It’s a description of all the functions, data types and data objects available to the application using the algorithm module.
  - Often, this is called an API (Application Programming Interface).

- When speaking abstractly (i.e. in general) about any algorithm module, XDAIS uses the term IMOD (short for MODule Interface).

- On the other hand, if you are describing a specific algorithm’s module, a unique interface name is used. For example:
  - If algorithm’s name is: FIR
  - We name its interface: IFIR

**Data Types**

typedef struct IFIR_Parms
typedef struct ...
IFIR_Parms myParms
IFIR_Object ...
int filter() ...

**Data Objects**

**Functions**

**IMOD**

**IFIR**

**Bottom Line**

- Think of a modules interface (i.e. IMOD) as its “prototype”
- For a module called FIR, its description is called IFIR

---

C6000 Integration Workshop - Using a XDAIS Algorithm
Basic XDAIS Components

1. Algorithm Parameters (Params)

- How can you adapt an algorithm to meet your needs?
  
  Vendor supplies “params” structure to allow user to describe any user-changeable algorithm parameters.

- For example, what parameters might you need for a FIR filter?

  *A filter called IFIR might have:

  ```
  typedef struct IFIR_Params {
      Int size;          // size of params
      XDAS_Int16 firLen;
      XDAS_Int16 blockSize;
      XDAS_Int16 *coeffPtr;
  } IFIR_Params;
  ```

2. XDAIS Components: Instance Object

- If you want to run the same algo on more than one channel...
  How do you prevent variables from conflicting with each other?

  *Each instance of an algorithm gets it’s own ‘storage’ location called an instance object.*

  **IFIR algorithm: Instance 1**

  - `*fxns` → Pointer to algo functions
  - `*a` → Pointer to coefficients
  - `*x` → Pointer to new data buffer

  **IFIR algorithm: Instance 2**

  - `*fxns`
  - `*a`
  - `*x`
3. XDAIS Components: Memory Table

- What prevents an algorithm from “taking” too much (critical) memory?
  - Algorithms cannot allocate memory.
  - Each block of memory required by algorithm is detailed in a Memory Table (memtab), then allocated by the Application.

- MemTab:

  ![MemTab Diagram]

  - **Size**
  - **Alignment**
  - **Space**
  - **Attributes**
  - **Base Addr**
  - **Space**: Internal / External memory
  - **Attributes**: Scratch or Persistent memory (discussed later)
  - **Base**: Starting address for block of memory

- MemTab example:

  ![MemTab Example Diagram]

  - **Application**
    - Based on the four memory details in MemTab,
    - Application allocates each memory block, and then
    - Provides base address to MemTab
  - **MemTab**
    - Size
    - Alignment
    - Space
    - Attributes
    - Base Addr
  - **Algorithm**
    - Algo provides info for each block of memory it needs,
    - Except base address …
XDAIS Example – Sinewave Algorithm

From the SINE.C code, it uses the following Data Elements and Functions

**Data** | **Scope**
--- | ---
FreqTone | global
FreqSampleRate | global
A | global
Y0 | global
Y1 | global
Y2 | global

**Functions**
- sineInit()
- sineValue()
- sineBlockFill()

SINE Example: Params & InstObj

1. Params

```c
typedef struct ISINE_Params {
    Int size;
    XDAS_Float32  FreqTone;
    XDAS_Float32  FreqSampleRate;
} ISINE_Params;
```

2. Instance Object

```c
typedef struct ISINE_Obj {
    struct ISINE_Fxns *fxns;
    XDAS_Float32  A;
    XDAS_Float32  Y0;
    XDAS_Float32  Y1;
    XDAS_Float32  Y2;
} ISINE_Obj;
```

And, the 3rd component we discussed?
**SINE Example: MemTab**

- How many blocks of memory does the Sine algorithm need?
  
  *Only one - for the Instance Object itself*

- The sine algorithm’s MemTab looks like:

  ```c
  IALG_MemRec memTab[1];
  int buffer0[5];
  
  memTab[0].size = 5;
  memTab[0].align = 1;
  memTab[0].space = Internal;
  memTab[0].attr = 0;
  memTab[0].base = buffer0;
  ```

  Note: If an algorithm needs additional memory block, such as data buffers, MemTab would need additional records: e.g. `memTab[2]`

---

**Application’s Code: Static Sine Example**

```c
// Initialization Code
ISINE_Params sineParams;

sineParams = ISINE_PARAMS;        // Most algos have a set of default params
sineParams.freqTone = 200;        // 200 Hz
sineParams.freqSampleRate = 48 * 1024; // 48 KHz

IALG_MemRec memTab[1];           // Create table of memory requirements.
int buffer0[5];                  // Reserve memory for instance object
memTab[0].base = buffer0;        // with 1st element pointing to object itself

ISINE_Handle sineHandle;         // Create handle to InstObj
sineHandle = memTab[0].base;     // Setup handle to InstObj
sineHandle->fxns = &SINE_TTO_ISINE; // Set pointer to algo functions

call sineInit                    // Exact syntax is shown later

// Runtime Processing

call sineValue                   // To generate a single sinewave value
```

- Star symbols indicate small amount of “extra” code required when using XDAIS
- Note, extra code only affects initialization of algorithm, **not runtime processing**
- This example uses “Static” allocation of memory in application code.
Algorithm Instance Lifecycle

Sine Algorithm Functions

- Once again, here are the functions from our Sine example:

<table>
<thead>
<tr>
<th>Sine Algorithm Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINE_init()</td>
</tr>
<tr>
<td>SINE_value()</td>
</tr>
<tr>
<td>SINE_blockFill()</td>
</tr>
</tbody>
</table>

Why did we group the functions as shown?

Algorithm Instance Lifecycle

- Once again, here are the functions from our Sine example:

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>SINE_init()</td>
</tr>
<tr>
<td>Process</td>
<td>SINE_value()</td>
</tr>
<tr>
<td></td>
<td>SINE_blockFill()</td>
</tr>
<tr>
<td>Delete</td>
<td>- none -</td>
</tr>
</tbody>
</table>

- SINE_init() initializes the memory used by the sine algo
- How was this memory allocated?
- In the last example, we did it statically:

```c
IALG_MemRec memTab[1];
int buf0[5];
memTab[0].base = buf0;
```

Can we dynamically instantiate an algorithm?
When dynamically instantiating an algorithm, a few more functions are required:

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>SINE_init</td>
<td>algNumAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algInit (aka sineInit)</td>
</tr>
<tr>
<td>Process</td>
<td>SINE_value</td>
<td>SINE_value</td>
</tr>
<tr>
<td></td>
<td>SINE_blockFill</td>
<td>SINE_blockFill</td>
</tr>
<tr>
<td>Delete</td>
<td>- none -</td>
<td>algFree</td>
</tr>
</tbody>
</table>

Notice the **additional functions**, Let's look at the process more closely...

Instance Creation - start

Application Framework

1. Here’s the way I want you to perform...
   
   Params = malloc(x);
   *Params = PARAMS;

Notice the use of dynamic memory allocation.
And the fact the algo never does the allocation.
1. Here’s the way I want you to perform…
   `Params = malloc(x);`
   `*Params = PARAMS;`

2. How many blocks of memory will you need to do this for me?

3. I’ll need “N” blocks of memory. (N may be based upon a params value)

4. I’ll make a place where you can tell me about your memory needs…
   `MemTab = malloc(5*N)`

5. Tell me about your memory requirements…

6. I’ll enter my needs for each of the N blocks of memory, given these parameters, into the MemTab…

7. I’ll go get/assign the memory you need…
   `for(i=0;i<=N;i++)`
   `mem = malloc(size);`

8. Prepare the new instance to run!

Now I can run the “processing” functions of the algo.
Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>algInit</td>
<td>algNumAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algInit</td>
</tr>
<tr>
<td>Process</td>
<td>SINE_value</td>
<td>SINE_value</td>
</tr>
<tr>
<td></td>
<td>SINE_blockFill</td>
<td>SINE_blockFill</td>
</tr>
<tr>
<td>Delete</td>
<td>- none</td>
<td>algFree</td>
</tr>
</tbody>
</table>

- If all algorithms must use these ‘create’ functions, couldn’t we simplify our application code?

Dynamic (top) vs Static (bottom)

1. \( n = \text{fxns->ialg.algNumAlloc}(); \)  
   \( \text{//Determine number of buffers required} \)
   \( \text{memTab} = \text{(IALG_MemRec *)malloc (n*sizeof(IALG_MemRec));} \)  
   \( \text{//Build the memTab} \)
   \( n = \text{fxns->ialg.algNumAlloc((IALG_Params *)params,&fxnsPtr,memTab);} \)  
   \( \text{//Inquire buffer needs from alg} \)

2. for (i = 0; i < n; ++i) {  
   \( \text{memTab[i].base = (Void *)memalign(memTab[i].alignment, memTab[i].size);} \)

3. \( \text{alg = (IALG_Handle)memTab[0].base;} \)  
   \( \text{//Set up handle and *fxns pointer} \)
   \( \text{alg->fxns = &fxns->ialg;} \)

4. \( \text{fxns->ialg.algInit(alg, memTab, NULL, (IALG_Params *)params);} \)  
   \( \text{// initialize instance object} \)

1. \( \text{IALG_MemRec memTab[1];} \)  
   \( \text{// Create table of memory requirements} \)
   \( \text{int buffer0[5];} \)  
   \( \text{// Reserve memory for instance object} \)

2. \( \text{memTab[0].base = buffer0;} \)  
   \( \text{// with 1st element pointing to object itself} \)

3. \( \text{ISINE_Handle sineHandle;} \)  
   \( \text{// Create handle to InstObj} \)
   \( \text{sineHandle = memTab[0].base;} \)  
   \( \text{// Setup handle to InstObj} \)
   \( \text{sineHandle->fxns = &SINE_TTO_ISINE;} \)  
   \( \text{// Set pointer to algo functions} \)

4. \( \text{sineHandle->fxns->ialg.algInit((IALG_Handle)sineHandle,memTab,NULL,(IALG_Params *)&sineParams);} \)

Luckily, though, you shouldn't have to write this code, because ...
A Generic Create Function

<table>
<thead>
<tr>
<th>Create Functions</th>
<th>Reference Framework</th>
<th>Purchased Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>algNumAlloc()</code></td>
<td>ALGRF_create()</td>
<td><code>FIR_create()</code></td>
</tr>
<tr>
<td><code>algAlloc()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>algInit()</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Common for all XDAIS compliant algo’s**
- **These functions specified by XDAIS algorithm standard**

- **One create function can instantiate any XDAIS algo**
- **ALGRF library provided in Reference Frameworks**
- **Reference Frameworks (RF) are discussed further in the next chapter**

- **Can be as simple as a single-line function**
  - which only calls ALGRF_create
- **Easier than using ALGRF_create; no complex C casting**
- **Optional function per XDAIS standard**
*** this page only appears to be blank…it’s really not ***
Lab 11 – Integrating an XDAIS algorithm

We’re going to add an algorithm to our existing audioapp code. This algorithm will filter out the sine wave that has been added to the music. In order to integrate this XDAIS algorithm we’ll need to do the following:

- Create a C file that will init and create an instance of the algorithm
- Modify our audioapp.c file to call that filter at the appropriate time

![Diagram]

- Add a xDAIS FIR Filter to system
- Use filter to eliminate sinewave from audio stream

**XDAIS Files**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR.H (Vendor May Provide)</td>
<td>Contains FIR_create &amp; FIR_delete functions</td>
</tr>
<tr>
<td></td>
<td>These are framework functions</td>
</tr>
<tr>
<td></td>
<td>Not required by algorithm standard (but usually provided)</td>
</tr>
<tr>
<td>FIR_TTO.H (Vendor Provides)</td>
<td>Only contains one item</td>
</tr>
<tr>
<td></td>
<td>Defines Global Symbol of vTab (table of functions)</td>
</tr>
<tr>
<td>FIR_TTO.L62 &amp; FIR_TTO.PDF (Vendor Provides)</td>
<td>Algorithm Library Archive &amp; Documentation</td>
</tr>
<tr>
<td>IFIR.H (Vendor Provides)</td>
<td>Define Module-specific Interfaces &amp; Structures</td>
</tr>
<tr>
<td></td>
<td>E.g. IFIR_Params, IFIR_Obj, IFIR_Handle typedef's</td>
</tr>
<tr>
<td>IFIR.C (Vendor Provides)</td>
<td>Default Values for IFIR_Params</td>
</tr>
<tr>
<td>IALG.H (TI provides)</td>
<td>Define Standard Interface Functions &amp; Data Types</td>
</tr>
</tbody>
</table>
Lab 11 Procedure

In this lab, we're going to add an XDAIS algorithm to filter out the sinewave. We're going to use a FIR module that has been written to use ALGRF to make our job a lot easier. We'll use a DIP switch to turn the filter on and off so that we can verify that it is working correctly.

Open Audioapp Project

1. Reset the DSK, start CCS and open audioapp.pjt

Examine and Edit xdais.c

2. Open xdais.c

Locate the file xdais.c in the \audioapp directory and add this file to your project. Once added, open it up for editing. This file is very similar to the other files that we have provided for you in this workshop. We're going to add the code to create two instances of a FIR filter to this file.
3. Examine \texttt{xdais.c}

Let’s take a look at this file from top to bottom. You’ll see:
- A place to put the header files for BIOS
- A place to put the header files for XDAIS
- The function prototypes
- Some declarations and a place for global variables
- One semi-empty function: \texttt{initAlgs()}. You will fill in this function with the code needed to create two instances of the FIR filter. Here is a summary of the code that you will write:
  - Create two global FIR\_Handle's, one for each channel
  - Create a local parameters structure
  - Fill the parameters structure with the default values
  - Change some parameters to meet our needs
  - Create two instance of the algorithm using \texttt{FIR\_create()}
  - Since \texttt{FIR\_create()} uses ALGRF, we need to set it up

\textbf{Set Up ALGRF}

The \texttt{FIR\_create()} function that we are going to use is really just a "wrapper" for calling \texttt{ALGRF\_create()}. The \texttt{FIR\_create()} wrapper takes care of a lot of casting and nasty C "stuff" that we just don’t want to have to deal with. \texttt{ALGRF\_create()} uses BIOS's MEM Memory Manager to allocate the memory needed by an algorithm. Since BIOS allows you to have multiple heaps, ALGRF leverages this capability to allow algorithms to use internal and external memory. To do this, ALGRF needs to be told which heaps to use.

4. \textbf{Inside xdais.c, add the following function call in initAlgs() to set up ALGRF's heaps}

Here is the code that you will need to add (below the definition of \texttt{firParams}):

\begin{verbatim}
ALGRF_setup(ISRAM, SDRAM); for the C6416 DSK

or

ALGRF_setup(IRAM, SDRAM); for the C6713 DSK
\end{verbatim}

\textbf{Note:} We currently have a heap allocated in each of these memories inside the .cdb file. BIOS allows you to name the heaps whatever you like. The names are declared as an enumeration, so we need to reference them as we have done at the top of xdais.c. This allows us to use the names ISRAM (or IRAM) and SDRAM directly.

5. \textbf{Create a SDRAM heap}

Since we are telling ALGRF to use the SDRAM heap, we need to create one. Open the .cdb file and go to the MEM-Memory Manager under the System folder. Right-click on the SDRAM Memory Segment and choose properties. Check the box titled "create a heap in this memory" to create the heap.
Create the FIR Instances

Now we will write the code to create an instance of the FIR Filter for each of our channels.

6. Create two global FIR_Handles in xdais.c
   We need one for each channel, left and right. Name them algFirL and algFirR (for example):
   
   ```c
   FIR_Handle algFirL;
   ```

7. Inspect the local FIR_Params structure inside initAlgs()
   This definition is placed above the call to ALGRF_setup(). Note the structure is named `firParams`.

8. Examine firParams
   Inside the initAlgs() function, inspect the firParams structure that contain the default parameters, FIR_PARAMS. You should see this below the call to ALGRF_setup().

   Also notice the following steps have been completed for you:
   - Coeff pointer element (coeffPtr) points to (short *)coeffs. (the coefficients are located in a header file that we will add later).
   - The filter length element of firParams (filterLen) is set to 345 (which is the number of coefficients that we have).
   - Frame length is set to BUFSIZE (this is the number of elements that we want to process each time we call the FIR Filter).

9. Create two instances of the FIR filter algorithm by calling FIR_create() twice
   Now that we’ve initialized the parameters we’ll want to create an instance of our filter using these parameters. We’ll do that with the `FIR_create()` function. Here is an example that creates the left channel instance:

   ```c
   algFirL = FIR_create(&FIR_TI_IFIR, &firParams);
   ```

   Add the code to create an instance of the algorithm for the right channel. None of the parameters need to change.

   `FIR_create()` calls ALGRF_create() and presents it all of the correct parameters with the correct types. The first argument to `FIR_create()` is a pointer to virtual table of the algorithm for which we want to create an instance. This table is defined in the library for the algorithm. For more information on the `FIR_TI_IFIR` function table look in the `fir_ti.h` header file.
10. Add #include statements for these header files

We also need to add #include statements for the following header files for XDAIS in xdais.c:
- "algrf.h" needed for the prototype of ALGRF_setup()
- "fir.h" needed for FIR module functions and types
- "fir_ti.h" defines the function table FIR_TTO_IFIR
- "200hz bandstop order 344.h" has our coefficients
- "audioappcfg.h" has the declarations: ISRAM, IRAM, SDRAM

11. Save xdais.c.

Modify main.c

12. Add a call to initAlgs() to main()

Open main.c and add a call to initAlgs() to main(). Call this function just before you call initMcBSP();

13. Include xdais.h in main.c

This file has the prototype for the initAlgs() function and external references to the handles that we will need.

Apply the FIR Filter to the Audio

We're finally going to get rid of that awful sine noise (without using the DIP switch to turn it off).

14. Use the FIR_apply() function to apply the FIR filter to the audio stream

Find the place in the processBuffer() function where the data is currently copied. Just above this, add two calls to FIR_apply() to filter the audio stream. FIR_apply() is another FIR module function that makes it easy to call the FIR filter in the xdais instance. The calls to FIR_apply should look something like this:

FIR_apply(Filter Handle, Source Buf Pointer, Destination Buf Pointer);

15. Use an if/else statement and a DIP switch to control when the filter is applied

Use DIP switch one on the DSK to turn the filter on and off. When the DIP switch is down, run the filter, when the DIP switch is up, do the copy as we have been doing.

16. Include fir.h in main.c

This file has the prototypes and type information (FIR_Handle) that we need to call FIR_apply().
Add Files to Project

We need to add some supporting files to our project.

17. Add fir.c and ifir.c to your project
These files are located in `c:\iw6000\xdais\algFIR`. Once you've added these files, go ahead and take a quick look at them.

18. Add ALGRF and FIR filter libraries to your project
These files are located in `c:\iw6000\xdais\lib`.
C6416 DSK users will need to add the `algrf.l64` library and the `fir_ti.l64` library.
C6713 DSK users will need to add the `algrf.l62` library and the `fir_ti.l62` library.
The `algrf.l6*` has the ALGRF module's code, and the `fir_ti.l6*` library has TI's implementation of the FIR module's code.

19. Add a new include path to your project
In order for CCS to find all of the new header files, we need to tell it where it can find them.
In your project build options, click the Preprocessor category. Add the following paths to the Include Search Path (don't forget the semicolons):

```
;c:\iw6000\xdais\include;c:\iw6000\xdais\algFIR
```
## Build and the Run program

### 20. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

<table>
<thead>
<tr>
<th>Source Files</th>
<th>Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>&lt;csl.h&gt;</td>
</tr>
<tr>
<td>edma.c</td>
<td>&lt;csl.h&gt;</td>
</tr>
<tr>
<td>xdais.c</td>
<td>&quot;algrf.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;edma.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;edma.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;edma.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6713.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6713.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6713_dip.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6416.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6416.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;dsk6416_dip.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;mcbsp.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;mcbsp.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;mcbsp.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;firl.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;firl.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;xdais.h&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;xdais.h&quot;</td>
</tr>
</tbody>
</table>

22. Load and Run

23. Verify Operation

You should be able to use DIP switch 0 to turn on the sine wave, then use DIP switch 1 to turn it back off (or really filter it out). Here's a summary of how the DIP switches are being used:

<table>
<thead>
<tr>
<th></th>
<th>Up</th>
<th>Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 0</td>
<td>No sine wave</td>
<td>Add sine wave</td>
</tr>
<tr>
<td>Switch 1</td>
<td>Filter disabled</td>
<td>Filter enabled</td>
</tr>
</tbody>
</table>

24. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp\*.*  TO  c:\iw6000\labs\lab11

25. When you’re done playing, halt the processor and close CCS

You’re done.
Additional Topics

XDAIS Rules and Guidelines

**XDAIS Documentation Rules**

Don't know how fast it runs ... or how much memory it uses.

*Strict rules on vendor-provided documentation (PDF file).*

**TMS320 DSP Algorithm Standard**

**MEMORY & PERFORMANCE CHARACTERIZATION**

<table>
<thead>
<tr>
<th>Module</th>
<th>Vendor</th>
<th>Variant</th>
<th>Architecture</th>
<th>Memory Model</th>
<th>Version</th>
<th>Doc Date</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>TTO</td>
<td>min</td>
<td>62</td>
<td>Little endian</td>
<td>none</td>
<td>04.15.2001</td>
<td>fir_company123_min.62</td>
</tr>
</tbody>
</table>

ROMable (Rule 5)

| Yes | No | X |

Tape (Rule 117)

<table>
<thead>
<tr>
<th>nameTab</th>
<th>Attribute</th>
<th>Size (bytes)</th>
<th>Align (MAUs)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Persist</td>
<td>20</td>
<td>4</td>
<td>External</td>
</tr>
<tr>
<td>1</td>
<td>Persist</td>
<td>2 * (addr_end - 1)</td>
<td>2</td>
<td>DARAIMO</td>
</tr>
</tbody>
</table>

Note: The unit for size is in 16-bit bytes and the unit for align is Maximum Addressable Units (MAUs).

**XDAIS File Naming Convention**

Will the function names conflict with other code in the system?

- **Algorithm must be C callable and re-entrant**
- **Strict naming rules virtually eliminate conflicts**
- **Similar rules exist for variable and function names**

```
fir_company123_min.64
firc_company123_max.h62
```

- **Algorithm Module Name**
- **Vendor Name**
- **Variant**

L: library
h: header
62: C62x/C67x
64: C64x
Overview of the XDAIS Rules

- General “Good Citizen” Software Coding Rules
  - C callable & Reentrant
  - Naming conventions enforced to avoid symbol clashes
  - No direct peripheral interface or memory allocation
  - Relocatable data and code in both static and dynamic systems
  - No thread scheduling nor any awareness of controlling app
  - Pure data transducer; cannot alter the DSP environment

- Standard Algorithm Interface defined by TI
  - Defines a memory management protocol between application and algorithm for all compliant algorithm modules

- Packaging Rules
  - All algorithms packaged and delivered in a consistent format

- Documentation Rules
  - Algorithms must provide basic memory and performance information to enable “apples to apples” comparisons and to aid system designers with algorithm integration

XDAIS Certification

Improved Software Reliability

- All third party compliant algorithms have been submitted to and passed a formal test
- TI oversees the test that is fully automated, error free, and unbiased
- TI is moving to release the test tool so that customers can self-check their own algorithms
- When an algorithm formally passes, the owner gains the right to use the compliant logo
XDAIS Third Party Support

Tools of the Trade

3rd Party XDIAS Compliant Algo’s

- > 650 companies in 3rd party network
- > 1000 algorithms from
- > 100 unique 3rd parties

Creating a XDAIS Algorithm with Component Wizard

Code Written by Component Wizard

The component has been generated for you and is located in the following directory.

Project: C:\MyProjects\FIR
*** wow…another piece of wasted real estate…***