Advanced_Features_of_C6Accel

^ Up to main **C6Accel_Reference_guide** Table of Contents

This arcticle is part of a collection of articles describing the C6Accel included in DaVinci/OMAPL/OMAP3 devices. To navigate to the main page for the C6Accel reference guide click on the link above.

Contents

- 1 Chaining calls to kernels in a single API call to C6ACCEL
    - 1.1 Design for the Codec-application interace of C6Accel
    - 1.2 Enabling Chaining of APIs in the ARM side application code
- 2 Integrating C6Accel in user defined codec server
- 3 Adding a new kernel to C6Accel and integrating with codec server
- 4 Return to Subsystem Documentation

Chaining calls to kernels in a single API call to C6ACCEL

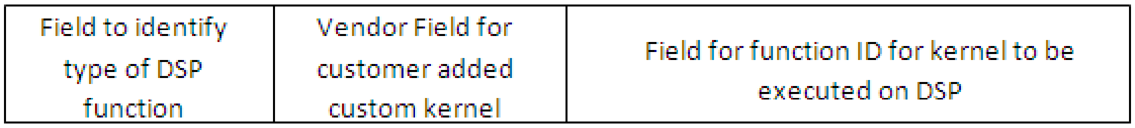
This is a feature of the C6ACCEL that allows the users to call multiple functions within the xdais algorithm using a single process call. This feature enables the application to save valuable overhead time introduced by making multiple process calls via the codec engine. However the use of this feature requires deeper understanding of the way the codec is instantiated from the application. Unlike the wrapper calls described in the previous section there are some memory management and alignment issues that the user is expected to know. Hence this can be seen as an advanced feature that the user can use at the cost of understanding the added complexity.

Let us begin with the understanding of what goes on inside the the wrapper code to understand how process calls are made to the kernels within the xdais algorithm. Some of the key application tasks managed by the wrapper functions are described in the Design for Codec-application interface of C6Accel section that follows.

Design for the Codec-application interace of C6Accel

The design section gives the user a peek into the design of the xdais source used to create the codec and also gives the user a sense of memory sharing issues faced in a DSP+ARM interfaced environment.

1. Function IDs: Each kernel within the xdais algorithm is assigned a unique function ID that the codec can recognize. The full list of function IDs can be seen in the header file iC6Accel_ti.h interface file residing in the base codec directory of C6ACCEL (~\ti\sdo\codecs\C6Accel\). Within each wrapper function the Function ID specific to the function gets passed as part of the input structure which is described in the next section. Format for the function IDs is given as



Field to identify type of DSP function	Vendor Field for customer added custom kernel	Field for function ID for kernel to be executed on DSP

The function ID will have three fields

- The first 8bit MSB field will identify the category of function being executed. (eg DSP ,Image ,Math ,Medical)
- The subsequent 8 bit field will be reserved for custom kernels that users might want to add to the XDAIS ALG. This field can be used as a vendor ID with the default vendor ID =0x01 reserved for TI.
- The 16 bit LSB will be used to identify the kernel that is to be executed on the DSP.

2. Extended Input arguments: The codec is built in a codec engine compliant IUNIVERSAL framework. The defualt input arguments for the IUniversal frame work limit the scope of passing all the parameter corresponding to a kernel. Hence in order to pass the user defined parameter set to the underlying functions we have extended the InArgs structure to the universal process call by defining an extended structure called IC6Accel_InArgs. This structure is quite similar to the inBufs,outBufs and inOutBufs in the IUniversal framework.

The extended input argument structure (IC6Accel_InArgs) enables the universal process call to extend the information carrying capacity of the inArgs. The extended structure would carry the information such as the number of functions being called in that particular API call and an array of function structures.

```
struct IC6ACCEL_InArgs{
                        Int size;
                        Int Num_fxns;
                        FXN_struct  fxn[MAX_FXN_CALLS];
                       }CODEC_InArgs;
```

Each function descriptor carrys the function id of the function to be executed and Parameter pointer offset that point to the memory location pointing to the parameters structure for that function.

```
     struct FXN_struct{
                      XDAS_Int32 Fxn_ID;
                      int Param_ptr_offset;
                     }FXN_struct;
```

Each function has its own Parameter structure defined in the C6ACCEL_ti.h interface header file.

```
        struct fxn1_Params{
                           XDAS_Int8 x_InArrID1;
                           XDAS_Int8 y_InArrID2;
                           XDAS_Int8 r_OutArrID1;
                           Int scalarParam1;
                           Int scalarParam2;
                          }fxn1_params;

        struct fxn2_Params{
                           XDAS_Int8 x_InArrID1;
                           XDAS_Int8 y_OutArrID1;
                           Int scalarParam1;
                          }fxn2_params;
```

The parameter pointer offset in each of the FXN_struct corresponding to the memory offset of the parameter structures from the start address of the contigous memory allocated to pass the parameters of the functions being called from C6accel.These offsets need to be pointed to its appropriate parameter structures. All buffers, vectors to be passed to the functions are passed using the inBufs and the outBufs of the Universal process API calls. Each function parameter structure contains IDs that are responsible to identify the correspondance between the inBufs and outBufs being passed and the function argument. The InArrIDs and outArrIDs are defined to accomplish this association.

For Example: If we make a call to a function DSP_Fxn(*x,*y,*r) where x,y are input vectors and r is the output vector such that we pass x_InArrID1= INBUF7, y_InArrayID2=INBUF8 and r_OutArrayID1=OUTBUF4 then C6Accel xdais algorithm will interpret this as x vector can be found in inBuf[7], y vector can be found in inBuf[8] and r output vector has to be passed using outBuf[4].

In addition to these IDs the function parameters also carry scalars to be passed to the underlying kernel being called.

3.Memory Management: The application code runs on the ARM while the codecs run on the DSP. Hence it is vital for the application to pass information to the codec which can be interpreted accurately by the DSP. The DSP and the ARM exchange information using a shared memory space in the DDR2 which is defined in the CMEM module. The ARM sees these locations as an virtual address through the MMU while the DSP sees it as the physical address. The codec engine performs address translation of the input, output buffers and its input/output arguments however, it does not translate the address for pointers being passed as a part of extended structures. This prevents application to directly pass pointers from ARM to DSP as part of Input arguments. Hence in order to pass multiple parameters we define the extended input Parameter set and allocate contiguous memory for these extended parameters. The codec engine invalidates and address translates input arguments of a iUniversal call based on the iUniversal_InArgs.size passed from the application.In C6accel we take advantage of this fact and ensure that all our input parameters are address translated by passing this size to be the size of the whole contiguous memory we allocate for the input arguments. (We have designed the CInArgs.size to be the first field in the extended structure to match the location of the IUniversal_InArg.size in the iUniversal InArgs structure.)

In addition to the memory translation issue, it is worth noting that DSP processes buffers and data aligned contiguosly in memory while the ARM has the capabilty to work on framented buffers because of its MMU. Hence it is important to pass buffers and parameter information which are aligned contiguously in memory. DMAI API Buffer_create() or Codec engine API Memory_alloc() [and Memory_contigalloc()]can be used to allocate contiguous memory buffers for function parameters from the CMEM module.

The wrapper library function calls hide these three complexities from the application developer and lets the application developer call into the kernels using API calls that look quiet similar to any function call made in C. However in order to call multiple functions within the codec, the application developer needs to handle these complexities.

Enabling Chaining of APIs in the ARM side application code

In the previous section we discussed the challenges of an SoC environment and discussed the C6Accel design to handle these issues. Now let us take a quick look at the implemention to chain API calls in C6Accel. In the application code, to call multiple functions in C6Accel, the application code would begin by reserving memory for the extended input arguments and the parameters to be passed by requesting contigous memory from the CMEM module. Let us take the case where the application wishes to call two functions using a single API call.The memory allocation call would appear as

```
IC6ACCEL_InArgs *CInArgs;
BASE_ADDR=(XDAS_Int8 *)Memory_contigalloc( MAX_FXN_CALLS*sizeof(FXN_struct)+ sizeof(fxn1_params)+sizeof(fxn2_params)+...+sizeof(fxnN_pa
```

The CInArgs argument to be passed to C6Accel is initialized to this base address

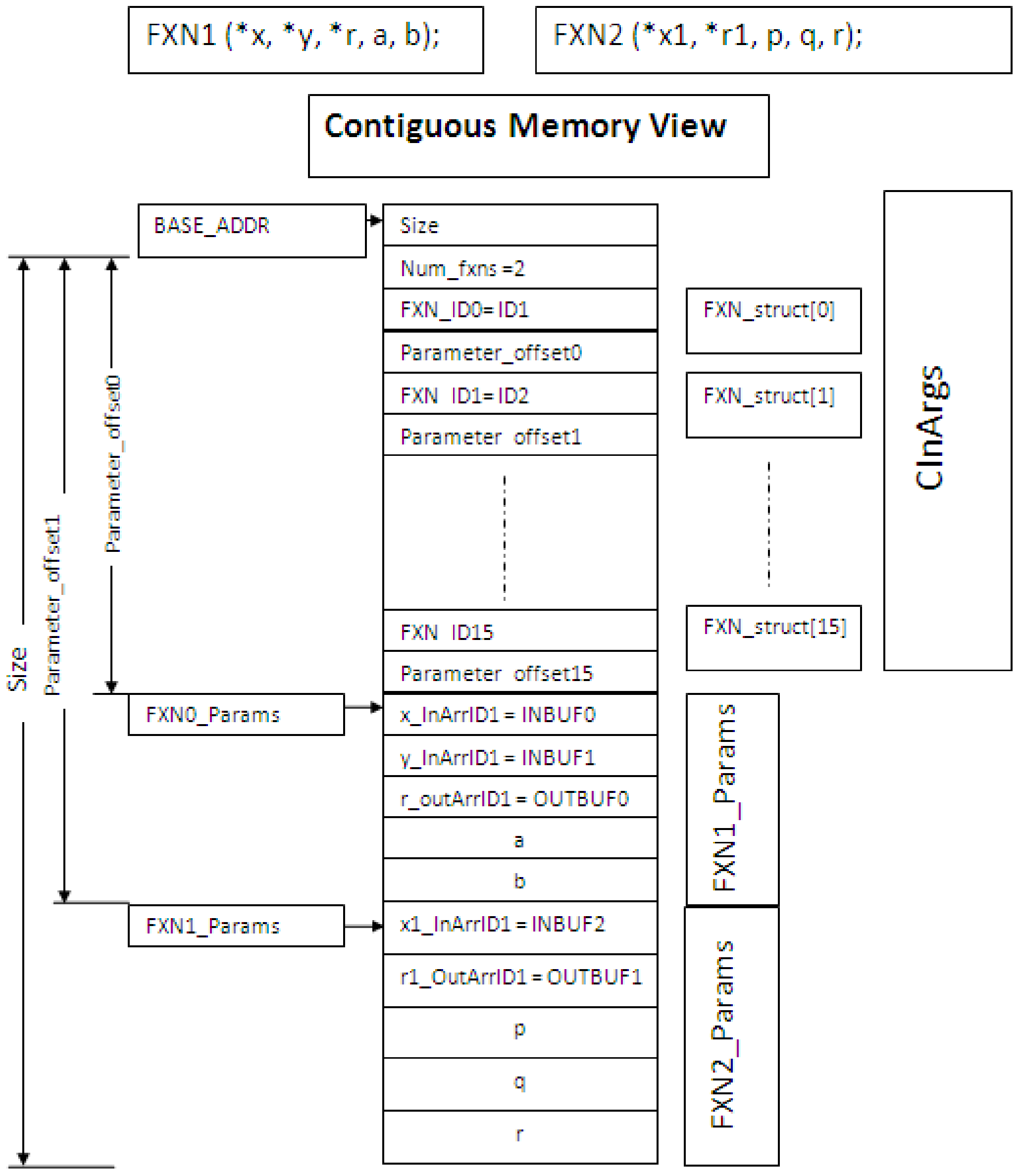
```
          CInArgs = (IC6Accel_InArgs *)BASE_ADDR;
```

The application code will pass the FXN_ids of DSP kernels being called. It would also have to compute the Parameter_ptr_offsets from the BASE address for every active element of the array of FXN_struct. This can be done by computing the offset of each parameter structure from the BASE address. The parameters are then initialized. This process would appear as defined in the example code below:

```
     //Initialize the extended InArgs structure //
```

```
        CInArgs->Num_fxns=2;
        CInArgs->size= sizeof(IC6Accel_InArgs);

      //Set function Id and parameter pointers for first function call
        CInArgs->fxn[0].ID= fxn1_ID;
        CInArgs->fxn[0].Param_ptr_offset = sizeof(Num_fxn)+sizeof(size)+MAX_FXN_CALLS*sizeof(FXN_struct);

      //Set function Id and parameter pointers for second function call
        CInArgs->fxn[1].ID= fxn2_ID;
        CInArgs->fxn[1].Param_ptr_offset = CInArgs->fxn[0].Param_ptr+sizeof(fxn1_params);

      //Initialize pointers to function parameters
        fp0= (fxn1_param *)(BASE_ADDR + CInArgs->fxn[0].Param_ptr);
        fp1= (fxn2_param *)(BASE_ADDR + CInArgs->fxn[1].Param_ptr);

      //Pass values to function1 parameters
        fp0->InArray1_ID = INBUF0;
        fp0->InArray2_ID = INBUF1;
        fp0->OutArray1_ID = OUTBUF0;
        fp0->scalarparam1 = INPUTHEIGHT;
        fp0->scalarparam2 = INPUTWIDTH;

      //Pass values to function2 parameters
        fp1->InArray1_ID = INBUF2;
        fp1->OutArray1_ID = OUTBUF1;
        fp1->scalarparam1 = INPUTHEIGHT*INPUTWIDTH;
```

The Contiguous Memory view of this is depicted in the figure below:



The codec engine invalidates and translates the address of input arguments InArgs of a iUniversal process call based on the iUniversal_InArgs.size passed from the application.In C6accel we take advantage of this fact and ensure that all our input parameters are address translated by passing this size to be the size of the whole contiguous memory we allocate for the input arguments. (We have designed the CInArgs.size to be the first field in the extended structure to match the location of the IUniversal_InArg.size in the iUniversal InArgs structure.)

```
CInArgs->size= InArg_Buf_size;
```

Once the address translation issue is addressed the application code can call into the codec using the universal_process call.

```
 //Make universal process call to the codec
   status = UNIVERSAL_process(hUni, &inBufDesc, &outBufDesc, NULL,(UNIVERSAL_InArg *)&IC6ACCEL_InArgs,&UniOutArgs);
```

Note: Cache Invalidation of the input and output buffers on the ARM side should also be handled by the application

Integrating C6Accel in user defined codec server

Note: Users who want to use just the codecs and who intend to use the codec server(prebuilt) in the DVSDK package do not have to rebuild the codec server. This section is useful for users who have their own codec server and want to integrate the C6ACCEL in to their servers.

Codec Engine can only have one server package opened at a time in an application and so the C6ACCEL must be added to the existing user server package that includes the video/audio decoders/encoders. The following reference guide describes how to add the a codec to an existing codec server.

http://processors.wiki.ti.com/index.php/Codec_Engine_GenServer_Wizard_FAQ

Adding a new kernel to C6Accel and integrating with codec server

Information to add a new user kernel and instructions to integrate into existing codec server is explained in this section

Adding user defined custom kernels to the codec requires user to add the DSP side algorithm to the source xdais algorithm that is provided with the package.A good starting point for a user to understand adding new kernels to review the Design_for_the_Codec-application_interace_of_C6Accel.

Adding new kernels to the C6Accel source xdais algorithm does not require any prior understanding of creating an xdais algorithm using the IUniversal codec engine interface. User can merely follow the set of instructions mentioned below and have their algorithm integrated into the existing codec.

Step 1. Define a unique function ID for the kernels being added. User must maintain consistancy with the Function ID format defined in the file $(XDAIS_INSTALL_DIR)/examples/C6Accel_alg/iC6Accel_ti.h. The xdais algorithm splits the xdais algorithm based on Vendor ID and further using the function classification ID. Within the xdais algorithm the function ID section the kernel is merely recognized using the last 16 bits of the function ID and hence the user is required to create an equivalent function ID in the file C6accel.h by simply masking the vendor and the function calssification ID by 0x00. For eg.

Function Id for the DSP_fft32x32 in iC6Accel is given defined as

```
 #define DSP_FFT32x32_FXN_ID  0x01010009
```

Equivalent function ID for DSP_fft32x32 in C6Accel.h is given as

```
  #define FFT32x32_FXN_ID  0x00000009
```

Note : iC6Accel_ti.h is an codec-app interface header file which is share information between the application and the codec while C6Accel.h is a header file only used by the xdais algorithm within the codec.

In Progress

Return to Subsystem Documentation

Click here.