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.

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.

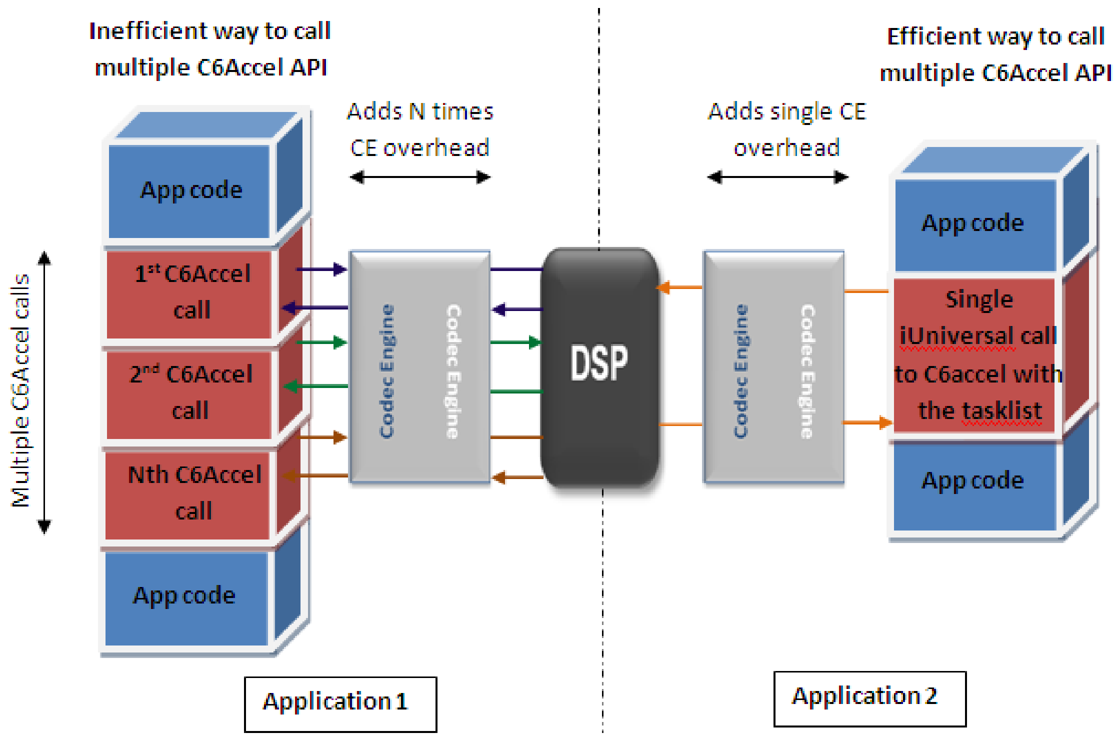


Figure: Block Diagram view of calling multiple DSP kernels in C6Accel

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_params)+
sizeof(CInArgs->size)+ sizeof(Num_fxns),BUFALIGN);
```

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

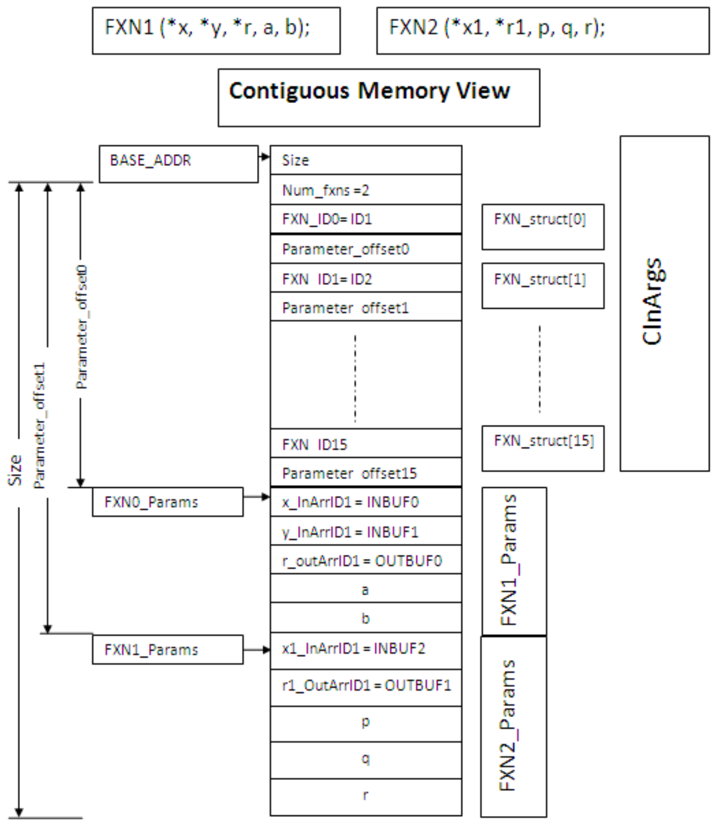
```
          CInArgs = (IC6Accel_InArgs *)BASE_ADDR;
```

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

Adding a new kernel/library to C6Accel

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

Users can add their own custom kernels to C6Accel 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 [1].

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. Add DSP kernel/Library to the source files

(a)Adding Kernel in source Add the source file for the custom DSP kernel to the $(C6ACCEL_INSTALL_DIR)/dsp/alg/src. For example refer to the complxtorealnimg in the above mentioned folder. Add the function definition to the C6accel.h file in the $(C6ACCEL_INSTALL_DIR)/dsp/alg/include file.

(b)Adding Kernel from a library To add a kernel from a library simply partially link the new library to the Archiver step in the Makefile found in $(C6ACCEL_INSTALL_DIR)/dsp/alg/pjt (or in the C6Accel.pjt if you are using CCS).

```
  LD_LIBS += -l $(LIBRARY_PATH)/USER_LIBRARY.lib
```

An example of doing this can be found the the C6Accel package. C6Accel links to a library called C64PLIBPLUS.lib(found under $(C6ACCEL_INSTALL_DIR)/dsp/libs) which contains additional DSP kernels that are not found in the standard C64P library offerings. This library is linked with the other libraries in the Makefile as follows

```
  LD_LIBS = -l"../../libs/dsplib64plus.lib"
 -l="../../libs/C64P_LIBPLUS.lib" -l="../../libs/fastrts64x.lib" -
l"../../libs/imglib2.l64P" -l"../../libs/IQmath_c64x+.lib"
-l"../../libs/IQmath_RAM_c64x+.lib"
```

Step 2. Define a unique function ID for the kernels being added.

User must maintain consistancy with the Function ID format defined in the file $(C6ACCEL_INSTALL_DIR)/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_complxtorealnimg in iC6Accel is given defined as

```
  #define DSPF_COMPLXTOREALNIMG_FXN_ID  0x01010410
```

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

```
   #define F_COMPLXTOREALNIMG_FXN_ID  0x00000410
```

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 xdais algorithm.

Step 3. Create Input parameter structure.

All functions in the xdais algorithm have their own input parameter stucture with varying number of fields depending on the number of arguments required for invoking the underlying DSP kernel. All input and output arrays are passed using inBuf and outBuf descripters of the iUniversal interface of the codec engine.Hence the input parameter structure only needs to carry the descripter ID corresponding to the input/output array as its field. All scalar arguments are passed as fields of the input Parameter structure. Example for creating a input structure for a DSP kernel is shown below.

Function call:

```
complextorealnimg( float *cplx, float *real, float *img, int nelements);
```

Input Parameter structure:

```
complxtorealnimg_Params={
                         unsigned int cplx_InArrID1;
                         unsigned int real_OutArrID1;
                         unsigned int img_OutArrID2;
                         int nelements;
                        }complxtorealnimg_Params;
```

Note: Ensure that the change made in iC6Accel_ti.h file is reflected in /soc/packages/ti/c6accel/iC6Accel_ti.h. Inorder to do this simply copy the file from the DSP path to its path in the SOC path in the package

Step 4. Insert case statement for the kernel.

Based on the function ID created insert a case statement inside the switch statement inside the appropriate Vendor ID and Library ID. The complextorealnimg has function ID which describes it as a DSP library function created by the TI as the vendor so we insert the function in

```
switch(FXN_ID&VENDOR_MASK>>VENDOR_SHIFT)
 case(TI):
      switch(FXN_ID&FXN_TYPE_MASK>>FXN_TYPE_SHIFT)
        case(DSPLIB):
            switch(FXN_ID&FXN_MASK)
              :
              :
              :
              case(COMPLXTOREALNIMG_FXN_ID):
               {
                /* Define pointer to parameter structure */
                   DSPF_complxtorealnimg_Params
*C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr;
                   C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr =
pFnArray;**
                /* Check parameters for restrictions */

if(((C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->cplx_InArrID1)>INBUF15)|

((C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->real_OutArrID1)>OUTBUF15)|

((C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->img_OutArrID2)>OUTBUF15)|

((C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->n) < 0)){
                        return(IUNIVERSAL_EPARAMFAIL);
                        }
                      else
                   /* Make function call*/
                      complxtorealnimg((float
*)inBufs->descs[C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->cplx_InArrID1].buf,
                                    (float
*)outBufs->descs[C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->real_OutArrID1].buf,
                                    (float
*)outBufs->descs[C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->img_OutArrID2].buf,

C6ACCEL_TI_DSPF_complxtorealnimg_paramPtr->n);
                  }
                 break;
          break;
          case(IMGLIB):
            :
          break;
          case(MATHLIB):
            :
          break;
   break;
```

****** pFnArray: pointer to function Array. This is computed before the start of the switch statement from the parameter offset field of the extended input arguments.

Step 5. Build the C6Accel library.

Once all the above mentioned code modification are completed, rebuild the package.In order to rebuild the package, execute the following commands from the root package directory $(C6ACCEL_INSTALL_DIR)

```
make clean
make all
```

The rebuild compiles the source files and archives the C6accel library files in the $(C6ACCEL_INSTALL_DIR)/soc/paclages/ti/c6acel/lib to be consumed by the application. When the codec server builds , it picks up this library, therby integrating the new kernel in the DSP executable that gets consumed in the application.

Step 6. Check for XDAIS compliance issues. It is a good practice to run the built C6Accel though the qualiTI tool to check for xDAIS complaince issues before integrating into a codec server. For more information refer to QualiTI [2].

Once the modified xdais algorithm passes the QualiTI test for xdais Complaince the modified C6Accel package is ready to be consumed in the codec server/unit servers.

Note: Step 7 is optional and must be followed only if developers wish to create an C6Accel API interface for the DSP kernel they have added to the xdais algorithm.

'Step 7.'**Create C6Accel ARM side API call.**

Source code for all C6Accel wrapper API calls can be found in $(C6ACCEL_INSTALL_DIR)/soc/c6accelw/c6accelw.c Create C6Accel API call for added kernel in the wrapper source file by adding the appropriate prefix (usually C6accel+Library name) to the kernel name. The API contains a C6accel handle argument in addition to all the arguments of the original DSP kernel call. All C6accel APIs return error codes so the return type is always integer value. For example C6accel API call for the complxtorealnimg

PSEUDOCODE

```
Int  C6accel_DSPF_complextorealnimg(C6accel_Handle hC6accel,float
*ptr_cplx, float *ptr_real, float *ptr_img,int npoints)  {
   /* Declare variable required to make iUniversal process call */
   XDM1_BufDesc                inBufDesc;
   XDM1_BufDesc                outBufDesc;
   XDAS_Int32                  InArg_Buf_size;
   IC6Accel_InArgs             *CInArgs;
   UNIVERSAL_OutArgs           uniOutArgs;
   Int status;

   /* Define pointer to function parameter structure */
   DSPF_complxtorealnimg_Params              *fp0;
   XDAS_Int8                   *pAlloc;

   /* This Macro assumes that the application prevents thread switch
from this point */
   ACQUIRE_CODEC_ENGINE;

   /* Allocate the InArgs structure as it varies in size (Needs to be
changed everytime we make a API call)*/
   InArg_Buf_size=  sizeof(Fxn_struct)+
                    sizeof(DSPF_complxtorealnimg_Params)+
                    sizeof(CInArgs->size)+
                    sizeof(CInArgs->Num_fxns);
   pAlloc=(XDAS_Int8 *)Memory_alloc(InArg_Buf_size,
&wrapperMemParams);
   CInArgs= (IC6Accel_InArgs *)pAlloc;

   /* Initialize .size fields for dummy input and output arguments */

   uniOutArgs.size = sizeof(uniOutArgs);

   /* Set up buffers to pass buffers in and out to alg  */
   inBufDesc.numBufs  = 1;
   outBufDesc.numBufs = 2;

   /* Fill in input/output buffer descriptor parameters (Macros Defined
 in c6accelw_i.h )*/
     /* These Macros are used to setup appropriate Buffer descriptor
for input/ouput arrays for the DSP kernel*/

CACHE_WB_INV_INPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_cplx,0,2*npoints*sizeof(float));

CACHE_INV_OUTPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_img,1,npoints*sizeof(float));

CACHE_INV_OUTPUT_BUFFERS_AND_SETUP_FOR_C6ACCEL(ptr_real,0,npoints*sizeof(float));
```

```
   /* Initialize the extended InArgs structure */
   CInArgs->Num_fxns = 1;
   CInArgs->size     = InArg_Buf_size; // Important step to
delegates memory handling to codec engine

   /* Set function Id and parameter pointers for first function call */
   CInArgs->fxn[0].FxnID     = DSPF_COMPLXTOREALNIMG_FXN_ID;
   CInArgs->fxn[0].Param_ptr_offset =
sizeof(CInArgs->size)+sizeof(CInArgs->Num_fxns)+sizeof(Fxn_struct);

   /* Initialize pointers to function parameters */
   fp0 = (DSPF_complxtorealnimg_Params *)((XDAS_Int8*)CInArgs +
CInArgs->fxn[0].Param_ptr_offset);

   /* Fill in the fields in the parameter structure */
   fp0->cplx_InArrID1   =  INBUF0;
   fp0->img_OutArrID2   =  OUTBUF1;
   fp0->real_OutArrID1  =  OUTBUF0;
   fp0->n               =  npoints;

   /* Call the actual algorithm */
   status = UNIVERSAL_process(hC6accel->hUni, &inBufDesc,
&outBufDesc, NULL,(UNIVERSAL_InArgs *)CInArgs, &uniOutArgs);

    /* Free the InArgs structure */
   Memory_free(pAlloc, InArg_Buf_size, &wrapperMemParams);

  /*Return Error code to the application */
   return status;

  /* This Macro informs the developer that the application can allow
thread switch from this point */
   RELEASE_CODEC_ENGINE;
 }
```

Step 8: Sanity Test for the kernel.

The C6Accel test application found in $(C6ACCEL_INSTALL_DIR)/soc/app is an test application which performs sanity checks on all the kernels in C6Accel. Once you rebuild the package after adding custom DSP kernel to the xdais algorithm, the kernel gets integrated into the unit server that can be found in $(C6ACCEL_INSTALL_DIR)/soc/packages/ti/c6accel_unitservers/$(PLATFORM). The test app is designed to consume this DSP executable. Hence in order to test the added kernel in C6Accel create test vectors for the kernel and make a call to the DSP kernel via codec engine using either the C6Accel API call or the iUniversal process call and check the output of the DSP kernel for correctness and performance.

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

Making Asynchronous Calls to DSP kernels in C6Accel

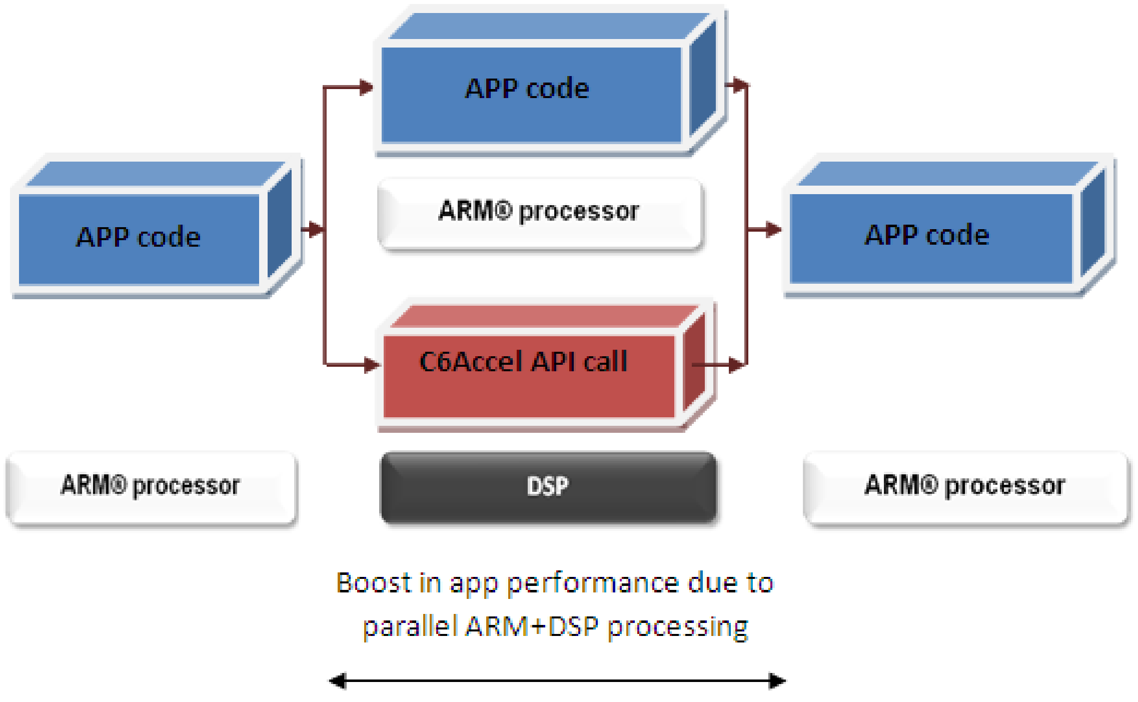


Figure: Application-Processor view for ARM application using C6Accel in Asynchronous mode

Asynchronous calling feature of C6Accel enables parallel processing on ARM and the DSP. Inorder to switch between Synchronous and Asynchronous calling C6Accel defines the following APIs

int C6Accel_setAsync(C6Accel_Handle hC6accel)

This sets calling mode to asynchronous.

int C6Accel_setSync(C6Accel_Handle hC6accel)

This sets calling mode to synchronous.

CALL_TYPE C6Accel_readCallType(C6Accel_Handle hC6accel)

This returns the current calling mode set in the application

Int C6accel_waitAsyncCall(C6accel_Handle hC6accel)

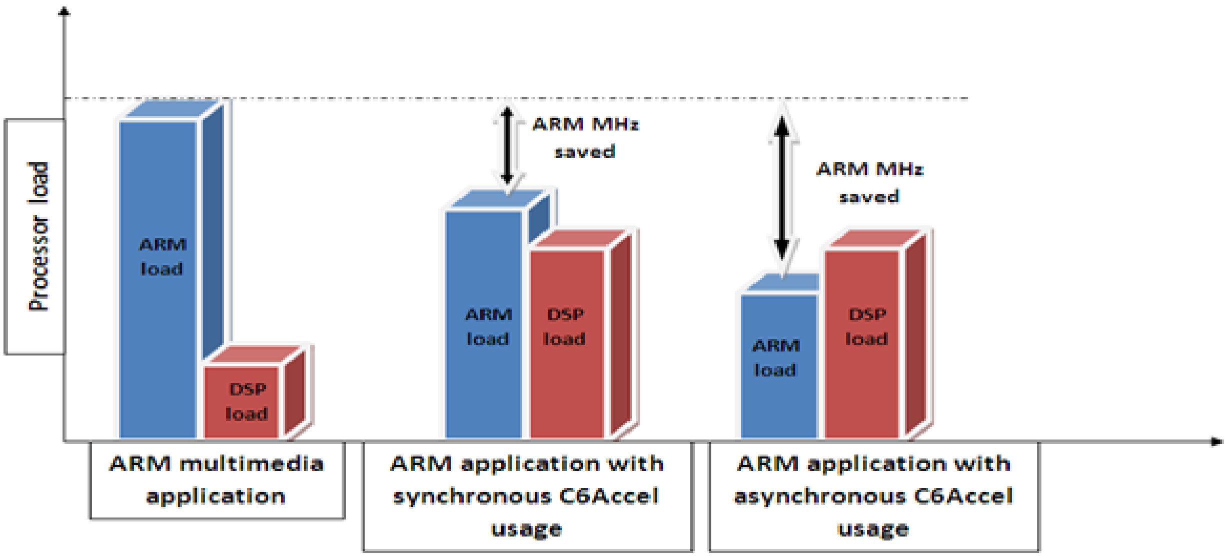
Wait for Async call to complete. The result from the DSP code will only be available when the Async call completes.

ARM application can make an async call and then perform other processing until the Async call completes, thereby allowing it maximum headroom for adding new features and improving performance.

Important Notes:

- APIs for calling DSP functionality do not vary in the Synchronous and Asynchronous mode.
- Application can make only one asynchronous call at a given time. In an asynchronous, the wrapper code does context saving for the call which is used in the waitAsyncCall.

The performance improvement obtained from Asynchronous processing is depicted below. The test application code included in the package contains example code to showcase this asynchronous calling.



Return to C6Accel Reference guide Documentation

Click here.

References

[1] http://ap-fpdsp-swapps.dal.design.ti.com/index.php/Advanced_Features_of_C6Accel#Design_for_the_Codec-application_interace_of_C6Accel
[2] http://processors.wiki.ti.com/index.php/QualiTI_XDAIS_Compliance_Tool

Article Sources and Contributors

Advanced Features of C6Accel *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=69004 *Contributors*: A0272049

Image Sources, Licenses and Contributors

Image:C6Accel_chaining_calls.bmp *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:C6Accel_chaining_calls.bmp *License*: unknown *Contributors*: A0272049
Image:Fxn_ID_update.bmp *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Fxn_ID_update.bmp *License*: unknown *Contributors*: A0272049
Image:Memcontig_view.bmp *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Memcontig_view.bmp *License*: unknown *Contributors*: A0272049
Image:C6Accel_async.bmp *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:C6Accel_async.bmp *License*: unknown *Contributors*: A0272049
Image:Async_call.bmp *Source*: http://ap-fpdsp-swapps.dal.design.ti.com/index.php?title=File:Async_call.bmp *License*: unknown *Contributors*: A0272049