Getting started with IUNIVERSAL

Getting started with IUNIVERSAL

- G Search for an article here:

Google™ Custom Search Search

Contents [hide]

Introduction

TI sells many system on chip (SoC) devices that feature both an ARM and a DSP. Typically TI provides an executable which runs on the DSP but is callable from the ARM to provide multimedia capabilities (e.g. H.264 encode/decode, etc.). TI customers frequently would like to have their own algorithm also capable of running on the DSP. Codec Engine provides a framework to enable customers to run their code on the DSP and to call it seamlessly from the ARM. The UNIVERSAL interface provides a method for customers to plug in any algorithm into this framework and take advantage of the dual core architecture.

This article intends to provide TI SoC users a process of writing algorithms which utilize the UNIVERSAL interface and Codec Engine framework to quickly get their code running on the DSP through function calls from the ARM.

Additional Background Info

[Skip this section if you're already familiar with VISA, XDM, XDAIS, RTSC.]

If you're not quite sure what this all means, let's take a few steps back: in the beginning, we wanted to be able to run multiple algorithms from multiple developers on a single DSP, so TI created a standard to ensure the algorithms didn't steal resources from each other, had unique namespaces, and could "play nice" together in general. There are a number of app notes and docs on this, so check out XDAIS Documentation. If you learn better by example, go to XDAIS Sample Algorithm. There's even a clever tool to make sure you've got it right: QualiTI.

Since most of the algorithms were strictly multimedia, TI then extended XDAIS to create a standard API (called XDM) specifically designed for multimedia codecs. These codecs are frequently referred to as VISA (for video, imaging, speech, and audio). Check out the XDM Users Guide. When combined with Codec Engine, we then have a nice way to simply "plug and play" our VISA codecs.

We then started getting questions like, "How can I get my face detection algorithm to work with Codec Engine?" One solution is to create adaptors as described here to make your algorithm look like a multimedia codec, and this worked well for a number of folks. But surely there should be another (possibly easier) way to get your algorithm to run – thus we get the introduction of the IUNIVERSAL API.

If you want a crash course on the above (XDAIS, XDM, RTSC Packaging, etc. but not IUNIVERSAL in particular), check out the freely available OMAP and DaVinci Software for Dummies book.

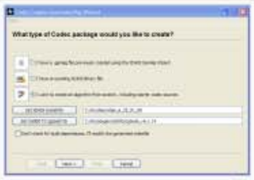
Procedure for Making DSP Algorithms that are callable from the ARM

Step 1: Invoke GenCodecPkg

The first thing to do is launch the wizard!

Step 2: Generate IUNIVERSAL starterware

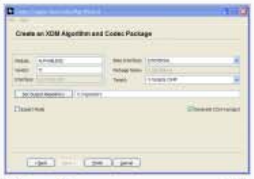
Screen 1



Screen 1

1. Click 3rd option, "I want to create an algorithm from scratch"
2. The XDAIS directory should already point to your XDAIS directory (frequently the $(CE_INSTALL_DIR)/cetools/packages directory if you're not using a DVSDK).
3. Point to the root of the compiler installation (i.e. one above the "bin" directory).
4. Click Next.

Screen 2



Screen 2

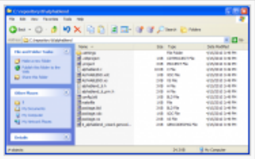
1. In the "Module" field enter the name of the algorithm you're creating.
2. In the "Vendor" field enter your company name.
3. Under "Base Interface" choose "IUNIVERSAL".
4. For "Target" choose C64P for 64x+ devices or choose C674 for 674x devices (floating point).
5. The output repository is where the generated files will be placed.
6. Click Finish to generate the starter files.

Generated Output

The generated files will reside at <output_repository>/<vendor>/<module.

Step 3: Build

Before making any changes, you should be able to build the generated package as is.


Output Files

Step 4: Customizing the Code to Fit Your Algorithm

Edits to <module>_<vendor>_priv.h

The object definition for `<MODULE>_<VENDOR>_Obj` is the **instance object** for your algorithm. The instance object is considered "private" in the sense that it is utilized internally by your algorithm. It should contain the **state** of each instance of your algorithm. In an FIR filter example this might consist of a pointer to the history buffer, a pointer to the filter coefficients, and the length of the filter. In this way each new filter that is created can have its own associated filter coefficients and history. Memory acquired during the **alloc** phase of the algorithm would likely have a parameter in this structure for use by the algorithm.

Edits to <module>_<vendor>.h

1. Define commands for use in the **control()** function.
   - Look for `#define I<MODULE>_USERCMD0`
   - Change `USERCMD0` to something more meaningful, e.g. `CHANGE_VOLUME`.
   - Remove the `#ifdef 0` from around the commands in order to use them.
2. Extend the `I<MODULE>_Params` as needed. This structure defines **only** parameters used at the time of instance creation. The `I<MODULE>_DynamicParams` structure we see later is what will be passed to the `control()` function.
3. Extend the `I<MODULE>_InArgs` structure as needed. These are the input arguments that will be passed to your `process()` function.
4. Extend the `I<MODULE>_OutArgs` structure as needed. This structure is used by the `process()` function to return info back to the application.
5. Extend the `I<MODULE>_DynamicParams` structure as need. This structure is passed along with a command to the `control()` function.
6. Extend the `I<MODULE>_Status` structure as necessary. This structure gets filled in by the `control()` function to provide status info back to the application.

Edits to <module>.c

1. `const I<MODULE>_Params I<MODULE>_PARAMS`
   - This code defines default parameters for instance creation.
   - When creating an algorithm instance the application has the option of passing a NULL handle for the `I<MODULE>_Params`. In that case the algorithm uses the default parameters which you define here.
2. `<MODULE>_<VENDOR>_alloc()`
   - This is the process by which you request memory for your algorithm. XDAIS algorithms never malloc() their own memory. They request it from the application through this function. This allows the system integrator to have better control over how memory is being given out.
   - By default it requests memory to contain your instance object. Any other memory needed (working buffers, etc.) should be requested here.
   - Each memTab entry tells size, alignment, internal/external, and persistent/scratch/write-once.
   - This function returns the number of entries it filled in. If you add any entries make sure you increment the return value accordingly.
3. `<MODULE>_<VENDOR>_free()`
   - Normally this can just utilize the `alloc()` function.
   - You only need to do something here if your initial memory request in `alloc()` was dependent upon the `I<MODULE>_Params` passed into the function.
4. `<MODULE>_<VENDOR_initObj()`
   - This function gets called after the application has granted memory requests based on your `alloc()` function.
   - The rest of the memTab has now been filled out by the application with the corresponding pointers to the memory you requested.
   - Those pointers should be assigned to corresponding members of your instance object for use during run-time.
5. `<MODULE>_<VENDOR>_process()`
   - This is where the actual algorithm goes! Add your algorithm code here.
   - **IMPORTANT**: For the purpose of cache coherence it's important to tell the framework about how the **CPU** (not DMA if you're using ACPY!) has accessed the buffers.
   - You must tell the application if you have read from or written to any of the buffers. This is achieved by using the `accessMask` fields associated with each and every buffer.
6. `<MODULE>_<VENDOR>_control()`
   - By default this function already supports the required command XDM_GETVERSION.
   - There is an `#ifdef 0` that you can get rid of to add handling for any commands that you defined in <module>_<vendor>.h.

Step 5: Creating a "server" that integrates your algorithm plus any others

Codec Engine supports running any IUNIVERSAL-compliant algorithm 'locally' (on the same processor as your app) or 'remotely' (on a slave processor, like the C64P of an OMAP3 device). See the Codec Engine GenServer Wizard FAQ for details on creating a server that integrates your GenCodecPkg-generated IUNIVERSAL algorithm into a DSP Server.

Step 6: Invoking the DSP Code from the ARM

- Tips for ARM side (Engine_open, etc.)

Step 7: Debug