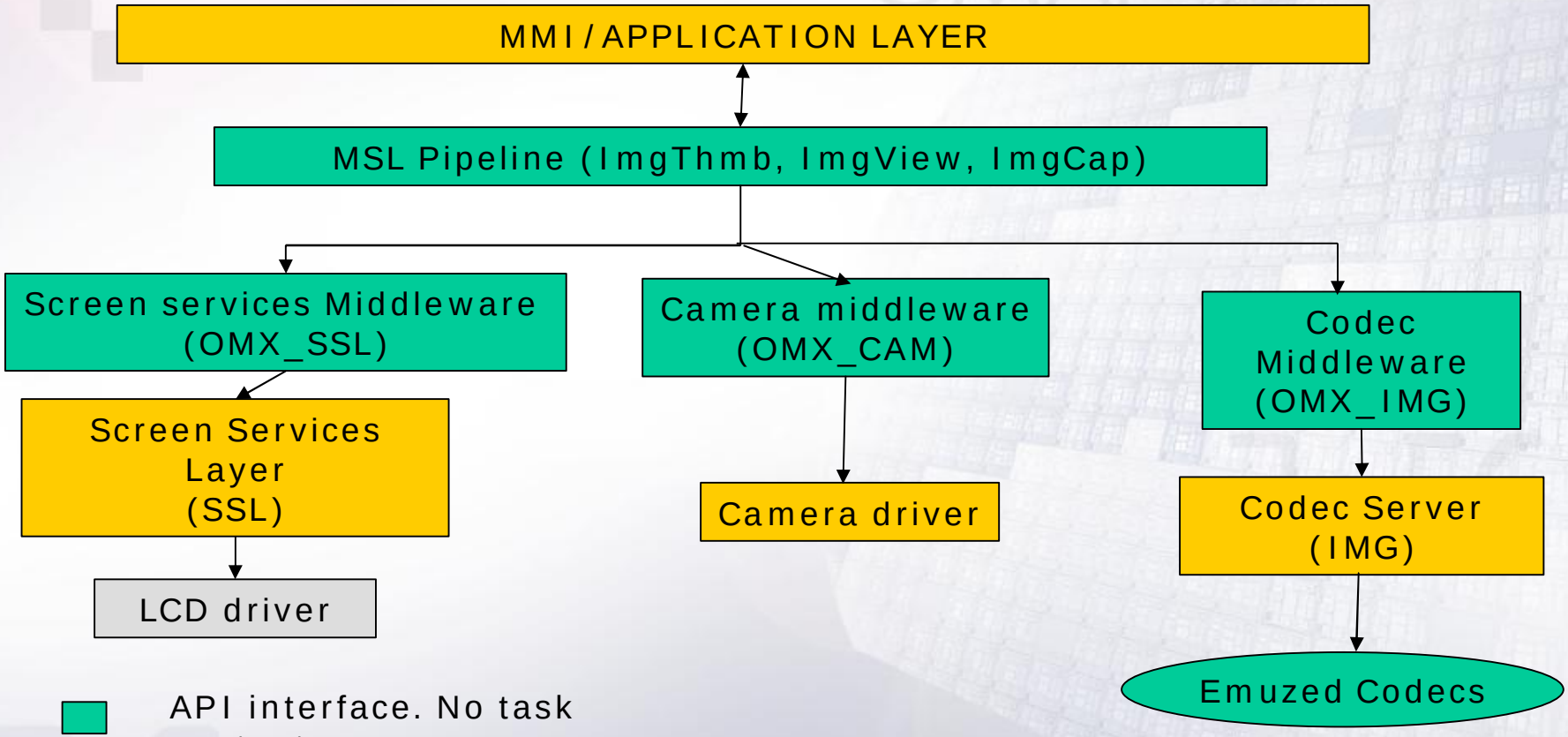
Locosto Multimedia Services Layer (MSL)

CSSD – MULTIMEDIA

What is MSL?

- Multimedia Service Layer (MSL) provides high-level APIs for applications to manage different use case pipeline.
- MSL abstracts internal complexities and provides a simple platform independent “c” API interface to applications.
- Regardless of the pipe line type, the API exposed is similar which enables customers to quickly incorporate a new pipelines.

Locosto MSL System Diagram



MSL Pipelines

- Current release provides MSL interface for camera and imaging applications.
  - Audio, Midi applications still use non-MSL older interface.

- Three pipelines are provided for imaging and camera applications
  1. MSLImgCap: This pipeline handles viewfinder application and image capture. This pipe line can be configured for different features like border frame, zoom, rotate etc for both viewfinder and image capture scenarios.

  2. MSLImgView: This pipeline handles image viewer scenario and image editing scenario. It supports decoding a jpeg image, do image processing operation over that like rotation, rescale/zoom, overlay etc and then finally display it to LCD using SSL.

  3. MSLImgThmb: This pipeline handles generation of thumbnail images. It is a generic pipeline which can decode JPEG files, rescale it down to user specified dimension and then re-encode. The use case envisioned for this pipe line is thumb nail image generation.

MSL APIs

- Each of the three pipelines expose following five APIs, which are common across all pipelines.

  1. MSL_<UCP>_Create (MSL_HANDLE *)
  2. MSL_<UCP>_Init (MSL_HANDLE)
  3. MSL_<UCP>_SetConfig(MSL_HANDLE, MSL_<UCP>_INDEXTYPES, MSL_VOID *)
  4. MSL_<UCP>_Deinit(MSL_HANDLE)
  5. MSL_<UCP>_Destroy (MSL_HANDLE)

  <UCP> - is the specific pipeline. It could be ImgCap, ImgView or ImgThmb.

- Apart from the above five, each of the pipelines expose one or two API for carrying out the core functionality. This is the only API which is specific to usecase pipeline. Even these APIs have same signature (takes same parameter)!
  - For e.g. ImgCap has an API MSL_ImgCap_Viewfinder(MSL_HANDLE hIMGCap); which starts the viewfinder.

MSL API details

- MSL_<UCP>_Create (MSL_HANDLE *);
  - This is the first API that application needs to call. Application should pass a **pointer** to MSL_HANDLE and this API will do all needed memory allocation and returns.
  - This is a synchronous API. i.e. no separate callbacks to indicate that functionality is complete.
  - The return status will indicate if the call was successful. It returns MSL_<UCP>_STATUS_OK. Most probably reason of unsuccessful return would be memory not sufficient.

MSL API details…

- MSL_<UCP>_SetConfig(MSL_HANDLE, MSL_<UCP>_INDEXTYPES, MSL_VOID *)
  - This API is used to set various MSL parameters. INDEXTYPES (second parameter) is used to indicate the parameter type. MSL will interpret the third parameter depending on INDEXTYPE value.
  - The index types are defined in the msl use case pipeline (ucp) header file (there is one header file per ucp).
  - There are two main INDEXTYPES, one used for initialization at the beginning for all parameters and the second index type for run time configuration of parameters. Run time parameter configuration would be like zoom change or rotate. This API could be called at any time.
  - This is a synchronous API. i.e. no separate callbacks to indicate that functionality is complete.

MSL API details…

- MSL_<UCP>_Init(MSL_HANDLE)
  - This API is used to initialize the MSL ucp pipeline.
  - This API should be called after the **first (only first)** MSL_<UCP>_SetParam API is called.
  - This is a asynchronous API. i.e. there will be a callbacks to indicate that functionality is complete. Note that a callback function of type MSL_CALLBACK will be called by MSL to indicate that initialization is complete. The callback API needs to implemented by application.
  - Application **should not** call any other MSL APIs until the callback function returns!

MSL API details…

- Process APIs
- ImgCap
  - MSL_ImgCap_Viewfinder(MSL_HANDLE hIMGCap)
    - This API will start the viewfinder application. This API after starting the viewfinder pipeline returns immediately to caller application. Viewfinder will keep running until a DEINIT API is called.
    - This API will call an asynchronous call back function MSL_CALLBACK once the initial viewfinder configuration is completed.
    - Different configuration for viewfinder like zoom, rescale, overlay needs to be set using setConfig API before calling this API
  - MSL_ImgCap_Snapshot(MSL_HANDLE hIMGCap);
    - This API will do the snapshot capture. After the init API either Snapshot or Viewfinder API can be called. This API can be configured to run in burst mode or single capture mode. This is decided by burst count parameter passed to setconfig API. Other configuration options are zoom, rotate and overlay.
    - This API will call an asynchronous call back function MSL_CALLBACK once the snapshot is completed. Snapshot involves capturing an image, do various image processing over that (rotate, overly, zoom etc) and then rescale it down and display a preview image.
- ImgView
  - MSL_ImgView_View(MSL_HANDLE hIMGView);
    - This API will start the image viewing pipeline. The image to be decoded and the post processing operation like overlay, rotate, zoom etc need to specific using a setcofnig API.
    - Once the image is decoded, post processed and display is completed it will make an asynchronous MSL_CALLBACK to indicate that the operation is completed.
- ImgThmb
  - MSL_ImgThmb_Generate(MSL_HANDLE hIMGThmb);
    - This API start the thumbnail generation pipeline. The original image, thmbnail dimension and thmbnail image name need to specific using a setcofnig API.
    - Once the thmbnail generation is over, it will make an asynchronous MSL_CALLBACK to indicate that the operation is completed.

MSL API details…

- MSL_<UCP>_DeInit(MSL_HANDLE);
  - This API is used to de-initialize the MSL ucp pipeline.
  - This API should be called before destroying an instance of MSL <UCP> pipeline. This API could be called at any instance. The normal use case is to call this application once the functionality is completed.
  - This is a asynchronous API. i.e. there will be a callbacks to indicate that functionality is complete. Note that a callback function of type MSL_CALLBACK will be called by MSL to indicate that de-initialization is complete. The callback API needs to implemented by application.
  - Application **should not** call any other MSL APIs until the callback function returns!

MSL API details

- MSL_<UCP>_Destroy (MSL_HANDLE *);
  - This is the last API that application needs to call. Application should pass MSL_HANDLE and this API will do de-allocation of MSL instance.
  - This is a synchronous API. i.e. no separate callbacks to indicate that functionality is complete. Once this function returns all memory allocated to the specific MSL instance will be deleted.
  - The return status will indicate if the call was successful. It returns MSL_<UCP>_STATUS_OK. Most probably reason of unsuccessful return would be memory de-allocation issues (which should not happen in a production system!).

Integrating MSL with applications

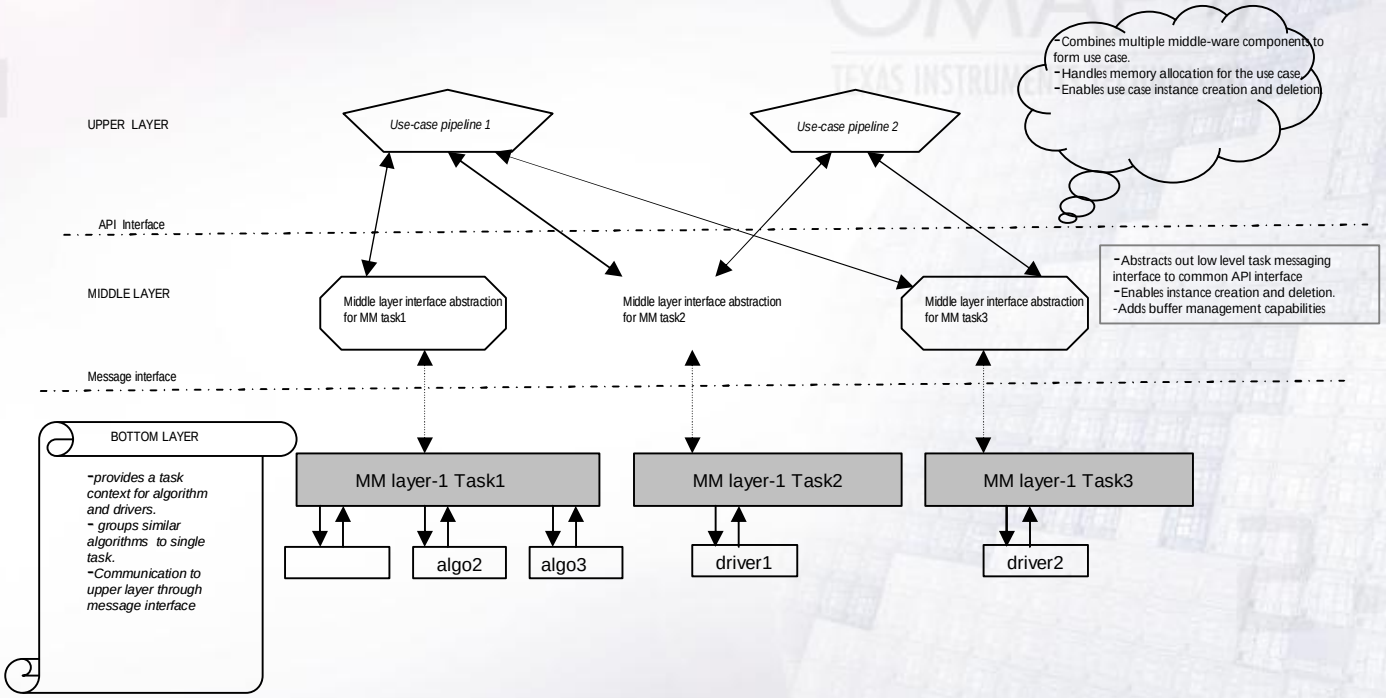
- MSL provides a top level header file, “msl_api.h” which defines all data types used across all msl pipelines. Applications should include this header file for using MSL layer.
- For each of the use case pipe lines a header file is provided and depending on the application this header file should be included. msl_imgcap.h, msl_imgthmb.h and msl_imgview.h are the three header files defined for currently defined pipelines.
- More detail documentation is provided in .\msl\docs\guides folder. Please do refer to .\msl\doc\release\cssd_relnotes_locosto_mm_msl.doc for change history.
- High-level estimate for integrating MSL is close to 1-week.

MSL ARCHITECTURE

MSL – DESIGN PHILOSOPHY

- This framework defines a complete layered architecture starting from codecs, all the way up to applications.
- It covers task models, component grouping strategies, memory allocation model and data pipelines.
- The framework deploys a novel statically configured *piggyback chaining approach* which enables simple component stacking for complex multimedia use cases
- The proposed multimedia framework is designed to be memory/performance efficient and statically scalable compared to other high-level multimedia frameworks where the focus is on generality and dynamic scalability.

MSL – framework layers



MSL – Bottom Layer

- The bottom layer constitutes of algorithms and drivers.
- This layer either generates data, consumes data or process data.
- This layer runs in its own task context. This layer does not deal with buffer handling or interfacing with other modules.
- This layer groups the data processing features. For e.g. all image processing algorithms are encapsulated as one single layer.
- This layer does not maintain states (pause, stop, play etc).
- The interface to this layer is always through message interface and status returns status to upper layer through callback APIs

MSL – Middle Layer

- The middle layer abstracts out creation, deletion and data passing mechanism to/from components.
- This layer provides mechanism for instances creation for a specific functionality. This layer maintains states.
- This layer provides an API interface (unlike bottom layer which provides message interface) and do not have a separate task context.
- The interface at this layer is common, regardless of underlying feature it abstracts out. The model that is followed in the proposed framework is very similar to industry standard OpenMax layer

MSL – Upper Layer

- The uppermost layer combines different middleware components to form final complex use cases. The use-case could be camera application, which combines capturing data from camera, pre-processing the data, encoding data to jpeg, generating preview image and displaying them on screen.

- This layer creates instances of middleware components, facilitates configuration of use-case features (like encode image dimension and quality, preview image dimension etc) and does the data chaining of these different middle layer components.

- This layer provides an API interface and do not have a separate task context. Regardless of the final usecase, this layer exposes a similar set of APIs, divided into synchronous and asynchronous set.

MSL – Event Driven Architecture

- The proposed multimedia framework is completely event driven.
- Middle layer carrier out the actual processing through the bottom layer, which runs from a separate task context. This ensures that upper layer is interrupted only when the required event processing is complete.
- Typical e.g. of events are, camera capture completed event by the driver or notification of image processing operation complete by the imaging task. These events are notified to top layer through a common set of middleware call back APIs.

MSL – Component Chaining

- In MSL model, the data pointers are passed from one component to another middle layer component in the first component's data callback function.

- To do away with the complexities of multiple callback function and their interaction, all middle-layer components for a specific use case (i.e. an instance of top layer) are configured to call the exact same callback API implemented in the upper layer.

- This API depending on the usecase, the configuration and state of the usecase (whether it is in the state of processing or in the state of stopping) decides whether and where to pass the data to the next component in the chain. This centralized callback/event manager brings in the simplicity and ability to add/remove a new component (or feature) from the usecase.

MSL – Camera Usecase study

- In the upper layer (to which the application layer communicates to), one instance each of omx_cam and omx_ssl is created. For each of the image processing requirement, a separate instance of OMX IMG module is created i.e. total of 4 instance (overlay, rotate, rescale/zoom, encode).
- The processing is initiated by the application by calling **MSL_ImgCap_Viewfinder** API.
- In the steady state, camera capture goes on in parallel with image processing (zoom, rotation, overlay etc) and data rendering to LCD.
- To enable this parallel processing of camera capture, image processing and data rendering, this use-case makes use of four buffers approach. This is done firing two buffers each to camera middle layer and to first img middle layer component in the chain. This ensures that after the first buffer is delivered to upper layer, post processed and delivered to ssl middle-layer, camera captures in parallel with image processing and display rendering.

MSL – Camera Usecase buffer flow