Vision SDK

Work Queue

User Guide

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards ought to be provided by the customer so as to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is neither responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

TABLE OF CONTENTS

1 Introduction

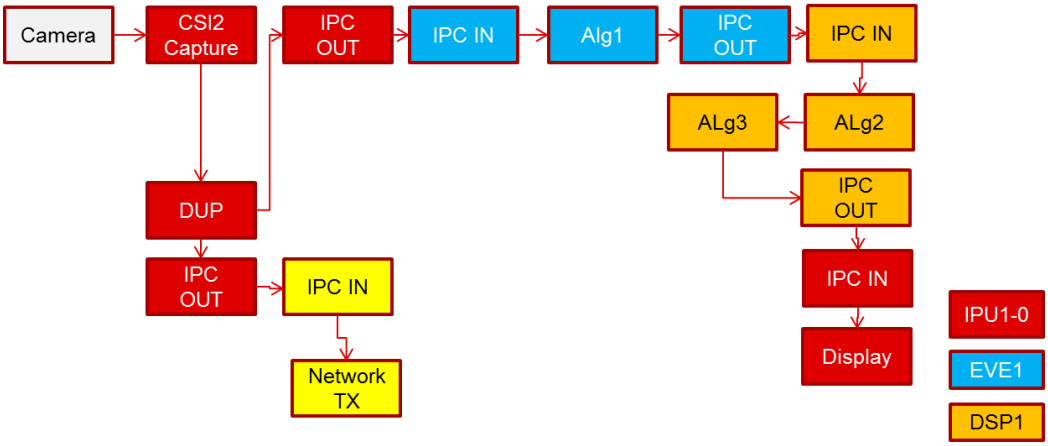
This document describes how to use the Work Queue (WorkQ) API in Vision SDK to offload work from a host processor to a remote processor.

WorkQ is a mechanism integrated in Vision SDK which allows a "host thread" running on a CPU like ARM M4 or DSP to offload "work functions" to a "worker thread" running on the same or remote CPU like EVE or DSP.

1.1 Motivation

The links framework in Vision SDK allows users to integrate algorithms and execute them with system level components. The links framework uses IPC IN and IPC OUT links to exchange data buffers between links running on different CPUs.

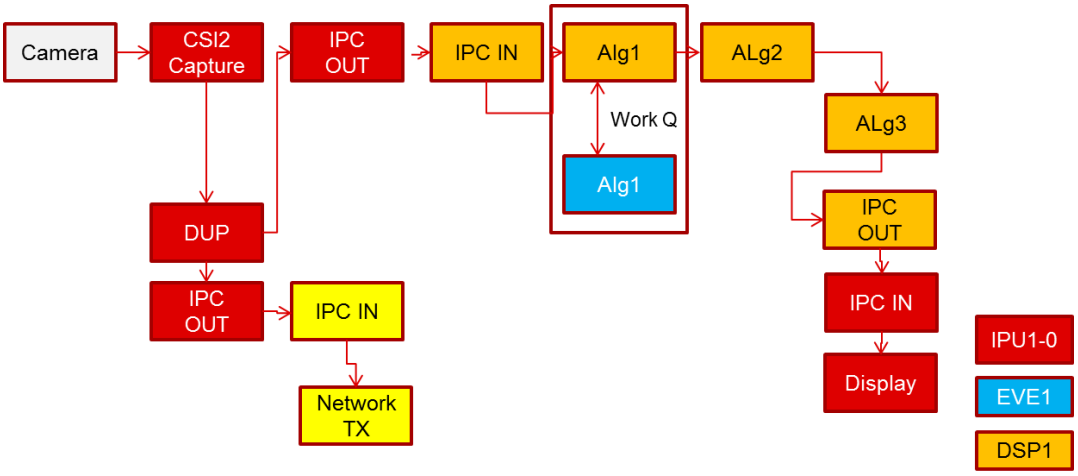
An example data flow using links framework is shown below,



Links framework offers flexibility where application developers can implement building blocks called "links" or "algplugins" and the Vision SDK allows these blocks to be connected to each other to form a data flow or a chain. "Links" framework takes care of control flow, pipelining buffers across links, inter processor communication.

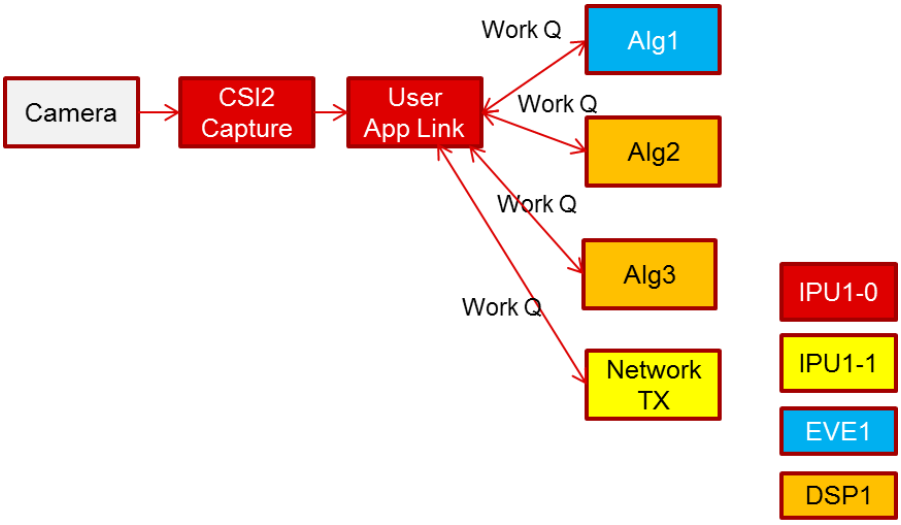
While links framework offers flexibility, sometimes however, a simpler "work offload" mechanism is desired where an application wants to execute a function on a remote CPU and it wants to do so in a quick and efficient manner. WorkQ provides such a mechanism. Inter processor communication is taken care by the WorkQ. However, when WorkQ is used, the application takes responsibility of system level control flow, data buffer pipelining.

An example data flow is shown below. Here WorkQ is used between DSP and EVE with DSP algplugin acting as the "host" thread and EVE running the "work function" thread. DSP algplugin submits work to EVE using WorkQ APIs rather than using IPC OUT, IPC IN and Algplugin links.



Multiple "Work functions" can be executed by the same WorkQ thread running on a remote CPU.

In most extreme case, "links", "algplugins" can be removed from all remote CPUs and only WorkQ based work functions can be integrated and executed by the application to realize the data flow. An example is shown below,

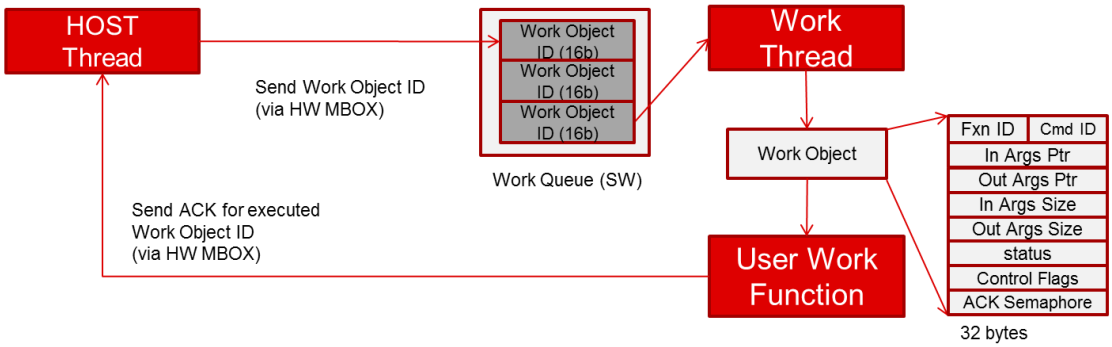


Here CSI2 Capture Link feeds frames to a User App Link. "Links" based framework data flow ends here. User App Link then uses WorkQ API to submit work to different CPUs and implements the control/data flow to execute Alg1, Alg2, Alg3, and Network TX across 3 different CPUs.

In general using WorkQ simplifies the algorithm integration logic on remote CPU. However it comes at cost of increased complexity and logic in the User App Link.

1.2 Software Block Diagram

The figure below shows the SW components involved during execution of WorkQ API.



1. HOST creates and setups "work object" once during init
2. At run-time only required fields updated by HOST
3. IPC involves sending a 16b work object ID via HW MBOX
4. Mailbox ISR queues Work IDs into a local SW queue (Work Queue)
5. Work Thread
   - dequeues "Work ID"
   - gets pointer to work object
   - executes user registered work function
6. After completion, ACK sent to HOST.
7. On receiving ACK, either user callback called or ACK semaphore posted

1.3 Features

- Supported on TDA2x, TDA2Ex, TDA3x SoCs running Vision SDK.
- Supported in BIOS ONLY configuration, i.e host thread and Work thread MUST run BIOS
  - **IMPORTANT NOTE:** Currently WorkQ is not supported in Linux+BIOS environment
- WorkQ Thread can be enabled on any of the supported CPUs like ARM M4, ARM A15, DSP, EVE
- HOST thread and Work thread can run on same CPU, i.e it is possible to submit work to Work thread running on same CPU
- HOST thread can be any normal OS thread including a "link" or "algplugin" thread.
- Multiple HOST threads on same or different CPUs can submit work to same or different Work threads simultaneously
- Work thread executes work in the order at which work arrives at the WorkQ
  - Work thread maintains a SW queue (FIFO) of pending work
- Work can be submitted via non-blocking APIs. This allows HOST to submit work and continue doing some operation before waiting for the Work completion status
- User can attach work completion callback or wait on work completion by calling a "wait complete" API
- User can trigger more work from within the completion callback function
  - This allows users to "chain" a sequence of work without switching to the HOST thread each time
- Users can register their own work functions and define their own parameters to pass to the work functions

2 Enabling WorkQ

1. WorkQ can be enabled by setting the below make variable as "yes" from within your make config file.
    - WORKQ_INCLUDE=yes
2. By default WORKQ_INCLUDE=no, after setting as "yes" in the cfg.mk file confirm the setting by doing a "make –s –j showconfig"
3. Re-build Vision SDK by following steps mentioned in SDK user guide.

2.1 Enabling WorkQ on EVE

- When WorkQ is enabled the task stack configuration in EVE is changed as below (\vision_sdk\apps\configs\<MAKECONFIG>/cfg.mk)
    - When WORKQ_INCLUDE=yes
        - systemTskMultiMbx uses EVE1_DATA_MEM, i.e DDR
        - workQueueTsk uses DMEM
    - When WORKQ_INCLUDE=no
        - systemTskMultiMbx uses DMEM
        - workQueueTsk is not created
- systemTskMultiMbx is the thread in which "links" framework runs algorithms
- workQueueTsk is thread in which WorkQ runs work functions (algorithms)
- It is not possible to run WorkQ functions and links framework in the same thread.
- For best EVE performance it is recommended that the thread which runs the algorithm uses stack located in DMEM
- However DMEM is not sufficient to hold stack for two threads.
- Hence for best performance on EVE, it is recommended that users either use WorkQ or links framework BUT not both
- NOTE: Links framework cannot be disabled as such via make option. Users should simply not create links/algplugins on EVE when WorkQ is enabled on EVE.
- There is no such constraint on other CPUs like M4, DSP, A15.

3 Using WorkQ

3.1 Important files

Description	File
User Interface header file	\vision_sdk\links_fw\include\link_api\system_work_queue_if.h
IPC Structure between HOST and Work Thread	\ vision_sdk\links_fw\include\link_api\system_work_queue_ipc_if.h
API documentation	\vision_sdk\docs\VisionSDK_ApiGuide.CHM [See Modules > Framework Module's > System Work Queue API]
Unit test case and example API usage	\vision_sdk\links_fw\src\rtos\links_common\system\ system_work_queue_unit_test.c
Implementation (HOST Thread)	system_work_queue_if.c
Implementation (Work Thread)	system_work_queue_tsk.c
Configuration parameters	system_work_queue_if_cfg.h
Implementation (private include file)	system_work_queue_if_priv.h

3.2 Basic Steps to use WorkQ

1. First step is to integrate a user function as a work function on the remote CPU running the work thread (section 3.3 Integrating User Work Function in Work Thread)
2. Next step is to call the previously registered user function from a HOST thread (section 3.4 Calling User Work Function from HOST Thread)

IMPORTANT NOTE:

- We use system_work_queue_unit_test.c as reference.
- In this section we focus on usage of WorkQ APIs. Refer VisionSDK_UserGuide_BuildSystem.pdf for details on how to add your own .c files to the Vision SDK build system.

3.3 Integrating User Work Function in Work Thread

1. Include the interface header file
   #include <include/link_api/system_work_queue_if.h>
2. Define a WorkQ function ID
   - #define SYSTEM_WORK_QUEUE_TEST_FXN_ID (0)
   - This is a direct index into a WorkQ function table for the given Work Thread
     - Hence make sure this is unique across the all Work function for a given Work thread.
     - Make sure this is less than SYSTEM_WORK_MAX_WORK_FXNS (system_work_queue_if_cfg.h)
3. Define a WorkQ command IDs for the work function
   - #define SYSTEM_WORK_QUEUE_TEST_CMD_ID (0)
   - This is a user defined 16b value and need not be unique across work functions.
   - Typically a user function would be an algorithm like say "Object Detect" and commands will be functions to be executed within the algorithm, ex, CREATE, RUN, DELETE, CONTROL etc
4. Define structure for input and output arguments
   - Typically each command will have different input and output arguments, thus one would normally define input and output structures for each command
   - Ex,

```
typedef struct {
   UInt32 value;
} System_WorkQueueTestFxnInArgs;

typedef struct {
   UInt32 value;
} System_WorkQueueTestFxnOutArgs;
```

   - Since there structures (or pointers to structure) can get passed across processors, make sure that fields used within these structure have a well defined size.
     - i.e avoid enum since size of enum can be different on different CPU depending on compiler options
     - Avoid using uint64_t since alignment of 64b integer on EVE and DSP/M4 is different
     - Avoid using bool since bool size could be compiler dependent
5. Implement the Work Function according to function signature as defined by typedef UInt32 (*System_WorkHandler)(

```
UInt16 workFxnId,  UInt16 workCmdId,
Void *pInArgs,  UInt32 inArgsSize,
Void *pOutArgs,  UInt32 outArgsSize);
```

   - This function is called by the WorkQ framework when it receives a work object with workFxnId as defined by user.
   - "workCmdId" is the command within the work function to execute

- pInArgs and pOutArgs are pointers to structure as defined by the user
- Typically there will be a switch case on the "workCmdId" within a work function and user will type cast to pInArgs and pOutArgs to the required user defined structures before using them
- In the example, the work function takes the value from input structure, increments by 1 and sets it to the value in output structure.
- "workCmdId" is not checked here since there is only one command associated with the "workFxnId"
- Return value for the work function is passed back to the caller on the HOST side. Framework does not use the return value.
- "workFxnId", "inArgsSize", "outArgsSize" can be checked for validity against user defined function ID and user defined structure size.
- Ex,

```
static UInt32 System_workQueueTestFxn(
            UInt16 workFxnId,  UInt16 workCmdId,
            Void *pInArgs,  UInt32 inArgsSize,
            Void *pOutArgs,   UInt32 outArgsSize)
{
   System_WorkQueueTestFxnInArgs *pIn;
   System_WorkQueueTestFxnInArgs *pOut;
   pIn = (System_WorkQueueTestFxnInArgs *)pInArgs;
   pOut = (System_WorkQueueTestFxnInArgs *)pOutArgs;
   pOut->value = pIn->value + 1;
   return (UInt32)0U;
}
```

- The memory pointed to by the pInArgs and pOutArgs could be cached. WorkQ framework takes care of doing appropriate cache operations before and after the work function is called.
- However if the user defined arguments itself point to additional memory then the user work functions MUST take care of cache operation for those memory pointers.
  - Ex, void *bufAddr can be a field within user input structure
  - User logic within the user work function MUST do appropriate cache operation before using data pointed by "bufAddr", ex, cache invalidate before read and cache writeback after write.

6. After the work function is implemented, register the work function against the user defined work function ID by calling below API
   - Ex,

```
Void System_workQueueTestFxnsRegister(void) {
    System_workHandlerRegister(
      SYSTEM_WORK_QUEUE_TEST_FXN_ID,
      System_workQueueTestFxn);
}
```

7. Compile the above work function code for the required CPU and call the register function during system init
   - Ex, call System_workQueueTestFxnsRegister inside AlgorithmLink_initAlgPlugins()

[\vision_sdk\links_fw\src\rtos\links_common\algorithm\
algorithmLink_cfg.c]

- Make sure to use appropriate #ifdef BUILD_<CPU> when calling the register API to ensure the work function gets linked against only the required CPU
- NOTE: for the unit test example System_workQueueTestFxnsRegister () is called by System_workInit() but for user functions it recommended to be called from within AlgorithmLink_initAlgPlugins()

3.4 Calling User Work Function from HOST Thread

3.4.1 Create Phase

1. Steps in this phase are typically done only during use-case create or system init and not for every work function execution.
2. Include the interface header file
   #include <include/link_api/system_work_queue_if.h>
   #include <utils_common/include/utils_mem_if.h>
3. Allocate a "work object" using the below API,
   System_WorkObjId workObjId;

   ```
   status = System_workAllocObj(&workObjId,
   SYSTEM_WORK_OBJ_ALLOC_FLAG_USE_ACK_SEM);
   UTILS_assert(status==SYSTEM_LINK_STATUS_SOK);
   ```

   - workObjId is used with all subsequent API to hold the context of the work that will be submitted.
   - SYSTEM_WORK_OBJ_ALLOC_FLAG_USE_ACK_SEM is used to tell the framework to allocate a semaphore. This will be used by the user via System_workWaitComplete() to wait for work function execution complete.
   - If user will not use System_workWaitComplete and instead use user callback for signaling work function execute complete, then this flag can be set to "0"
4. Associate a work function ID and command ID with the work object
   System_workSetCommand(workObjId,
   SYSTEM_WORK_QUEUE_TEST_FXN_ID,
   SYSTEM_WORK_QUEUE_TEST_CMD_ID);
   - The work function registered against the ID SYSTEM_WORK_QUEUE_TEST_FXN_ID, will be executed when the work object is submitted
5. Allocate input and output argument structures. Since the structure could be sent across CPUs, make sure to allocate them on cache line boundary. Use the macro SYSTEM_WORK_ARGS_ALIGN to align the allocation of these structures. Utils_memAlloc() can be used to allocate the input/ouput structures.
   - Ex,

     ```
     System_WorkQueueTestFxnInArgs *pInArgs;
     System_WorkQueueTestFxnOutArgs *pOutArgs;
     UInt32 inArgsSize;
     UInt32 outArgsSize;
     ```

```
    inArgsSize =
SystemUtils_align(sizeof(System_WorkQueueTestFxnInArgs),
SYSTEM_WORK_ARGS_ALIGN);

    outArgsSize =
SystemUtils_align(sizeof(System_WorkQueueTestFxnOutArgs),
SYSTEM_WORK_ARGS_ALIGN);

    pInArgs = Utils_memAlloc(UTILS_HEAPID_DDR_CACHED_SR,
inArgsSize, SYSTEM_WORK_ARGS_ALIGN);
    UTILS_assert(pInArgs!=NULL);

    pOutArgs = Utils_memAlloc(UTILS_HEAPID_DDR_CACHED_SR,
outArgsSize, SYSTEM_WORK_ARGS_ALIGN);
    UTILS_assert(pOutArgs!=NULL);
```

6. Associated the arguments with the work object that will be used to submit the work.
    - Ex,
      System_workSetInArgs(workObjId, pInArgs, inArgsSize);
      System_workSetOutArgs(workObjId, pOutArgs, outArgsSize);
7. Implemented user callback. This is optional. In the example, the user context which is a uint32 value is simply incremented.
   Ex,

```
static Void System_workQueueTestCallback(
                    System_WorkObjId workObjId, Void *pUserObj)
{
  UInt32 *pUInt32 = (UInt32 *)pUserObj;
  (*pUInt32)++;
}
```

8. If user callback is implemented thensSet user callback to invoke when ACK is receive for a submit work function
    - Ex, System_workSetCallback(workObjId, System_workQueueTestCallback, (Void*)&workQueueTestFlag);
    - &workQueueTestFlag is a optional user application context that can be passed to the API and that is made available when user callback is invokved
9. This completes the create phase

3.4.2 Execute Phase

1. In this phase the work object is submitted repeatedly to execute the work on the remote CPU or Work thread
2. Set the inputs arguments required by setting fields in the input, output structures
    - Ex, pInArgs->value = 0;
3. Submit the work,
    - Ex, System_workSubmit(workObjId, dstCpuId);
    - Here "dstCpuId" is the CPU on which to execute the work

- "dstCpuId" can be one of the valid values from SYSTEM_PROC_<CPU>
- "dstCPuId" can be same as current CPU

4. The System_workSubmit returns immediately after the workObjectId is put in the HW Mailbox for IPC (or SW Queue in case dstCpuId is same as current CPU ID)
5. The HOST thread can then continue with other work (including submitting additional work to other CPUs) OR wait for current submitted work to complete
6. If SYSTEM_WORK_OBJ_ALLOC_FLAG_USE_ACK_SEM was used during work object create then below API can be used to wait for work execution complete
   System_workWaitComplete(
   workObjId, SYSTEM_WORK_TIMEOUT_FOREVER);
7. If user callback is used then user callback can signal (via semaphore or other means) the work completion.
   - System_workSubmit() can be called from within user callback to chain a sequence of work without switching to the HOST thread.
   - NOTE: User callback is called in ISR context in BIOS and normal rules that apply to ISR should be followed by the callback implementation.
8. After work execution is completed, output arguments can be checked by the user and output status can be checked.
   - Output status is checked via the API
     Int32 System_workGetStatus(System_WorkObjId workObjId, UInt32 *workStatus);
9. The steps in execute phase can be repeated multiple times.
   - Users can create multiple work object created, and submit for execution at given point of time.
   - In the most simplest case, user can submit work and immediately wait for its completion.
   - There can more complex user logic to build a multiple work object submit state machine, which pipelines work across multiple CPUs to keep them busy and executing in parallel. The non-blocking WorkQ Submit APIs with user callback supports such a scheme but its upto the user application to implement this logic and not in scope of the WorkQ framework.

3.4.3 Delete Phase

1. This is done during use-case delete or system shutdown. In this phase memory and object allocated during create are released.
2. Free work object ID
   - Ex, System_workFreeObj(&workObjId);
3. Free input, output argument structures
   - Ex,
     Utils_memFree(UTILS_HEAPID_DDR_CACHED_SR, pInArgs, inArgsSize);
     Utils_memFree(UTILS_HEAPID_DDR_CACHED_SR, pOutArgs, outArgsSize);

4 Configuring WorkQ

WorkQ by default is configured for typical system use-cases, however users may want to change some parameters based on their specific required.

The important WorkQ configuration parameters which can be changed by users are defined in the file system_work_queue_if_cfg.h [\vision_sdk\links_fw\src\rtos\links_common\system]

Below table explains the configuration parameters available

Parameter	Description
SYSTEM_WORK_MAX_WORK_FXNS	Maximum WorkQ functions that can be registered against a given WorkQ thread.
SYSTEM_WORK_MAX_QUEUE_ELEMENTS	Maximum WorkQ functions that can be pending for execution at any given point in the SW Work Queue. i.e if HOST thread submit more work than this value then work could get dropped by the WorkQ function. NOTE: This would indicate a system design issue and solution would be to analyze how the work is being submitted by HOST threads and adjust this value accordingly. Each queue element size is 4B so it's not very expensive to increase this value.
SYSTEM_WORK_TSK_STACK_SIZE	Work Thread Stack size. By default all CPUs are set to have same stack size. Use $ifdef BUILD_<CPU> is different CPUs need to have different stack size. Typically stack for EVE is kept in DMEM so this value cannot be very large for EVE (typically 8KB is the recommended stack size for EVE)
SYSTEM_WORK_TSK_PRI	Work Thread task priority.

5 Inter Processor Communication in WorkQ

NOTE: This section can be skipped by users of WorkQ API. This section is for advanced users for informational purposes.

5.1 Basic IPC Mechanism used in WorkQ

This section describes the IPC mechanism used by the WorkQ framework.

- The WorkQ framework uses HW Mailbox to exchange work objects between CPUs
- A work object is defined by the below structure

```
typedef struct {
    UInt32 fxnIdCmdId;
    UInt32 flags;
    UInt32 *pInArgs;
    UInt32 inArgsSize;
    UInt32 *pOutArgs;
    UInt32 outArgsSize;
    UInt32 status;
    Void   *pSemaphore;
    System_WorkUserCallback userCallback;
    Void   *pUserObj;
} System_Work;
```

  - Typically user of WorkQ will set fields in the work object structure via WorkQ APIs only.
- A shared memory (in non-cached DDR section) maintains the work object structures
  - An array of Work Object structure is created for every possible CPU that can submit work

```
typedef struct {
    System_Work obj
            [SYSTEM_PROC_MAX]
            [SYSTEM_WORK_MAX_OBJ_PER_CPU];
} System_WorkIpcObj;
```

- A Work Object ID is a integer which is combination of the SRC CPU ID and Work Object array index from the shared memory data structure
  - Thus given a work object any CPU can dereference and get the work object structure information.
- Thus the unit of exchange across CPUs via the HW Mailbox is the Work Object ID
- This keeps the IPC extremely low overhead
- Work object ID is inserted in the HW mail box, an ISR is triggered on the DST CPU ID.
  - In the DST CPU ISR, the Work object is de-queued from the HW Mailbox and queue to a SW queue.
  - This allows users to queue work much more that what is allowed by the HW Queue within the HW Mailbox
- When work is submitted and DST CPU ID and SRC CPU ID are same then IPC is bypassed and work object ID is directly queued to the SW queue associated with the destination work thread.

5.2 Error condition handling during IPC

5.2.1 HW Mailbox getting full at Source CPU

- The HW Mailbox ISR for DST CPU de-queues from HW Mailbox and puts in SW queue.
- Hence unless the DST CPU has crashed, from the SRC CPU point of view a message put into the HW Mailbox will always be successful.
- There could be small momentary duration during which the SRC CPU submitted may have submitted work object IDs back to back even before ISR on DST CPU got a chance to execute. In this case once HW Mailbox is full, SRC CPU will poll until there is space in the HW Mailbox to accommodate new work object IDs.
    - Thus unless ISR on DST CPU does not executed, space will eventually be created in the HW Mailbox and the Work Object will get queued to the HW Mailbox

5.2.2 SW Queue getting full at Destination CPU

- Once a work object ID is de-queued from HW Mailbox it is queued to the SW Queue on the DST CPU. Typically the size of the SW Queue will be much larger than the HW Mailbox queue
- However if user application is done designed correctly then a condition could happen that the SW queue get full and new Work Object ID cannot be queued into it.
    - This can happen, when the rate at which work is submitted is higher than the rate at which work is being executed.
    - This will result in a work object remaining in SW queue pending to be executed. In this number is more than queue depth then SW queue will occur.
    - This is a system design problem and there can be no graceful recovery to this situation.
    - It should be solved by adjusting the work submit rate and/or increasing the SW queue size to required length.

6 Revision History

Version	Date	Revision History
0.1	03rd March 2016	Draft
0.2	29th June 2017	Updated for Vision SDK rel 3.0