Open Event Machine library

User Guide

Applies to Product Release: 01.06.00.02:
Publication Date: September, 2013

Contributors to this document

Texas Instruments, Incorporated
821 avenue Jack Kilby
06270 Villeneuve-Loubet Cedex,
FRANCE

TEXAS INSTRUMENTS

Revision Record	
Document Title: **User Guide**	
Revision	Description of Change.
00.00.00.01	Document created.
01.00.00.00	Public delivery.
01.00.00.01	Updated scope section.
01.00.00.02	Updated document with new OpenEM directory tree.
01.01.00.00	Updated document with information related to: - the new sections organization - the private event free queue - the extra event descriptor dedicated to OpenEM usage
01.06.00.02	Reworked document to integrate ARM and ARM & DSP sections. Those sections still remain to be completed. Aligned document on OpenEM changes.

TABLE OF CONTENTS

1 Acronyms

Acronym	Description
API	Application Programming Interface
CCS	Code Composer Studio
CPPI	Common Packet Programming Interface
DMA	Direct Memory Access
DSP	Digital Signal Processor
EO	Execution Object
EQ	Event Queue
OpenEM	Open Event Machine
PDSP	Packed Data Structure Processor
PKTDMA	PacKeT DMA
QMGR	Queue ManaGeR
QMSS	Queue Manager Sub-System
RTSC	Real Time Software Component

2 Conventions

- “shall” (“must”, “needs”) is used to express an obligation.
- “should” is used to express a recommendation.

3 Scope

This document addresses release 1.6.0.2 and later of the Open Event Machine (OpenEM).

4 Introduction

Purpose of the user's guide is to provide examples of OpenEM usage for both DSP and ARM.

All examples implement variants of an application parallelizing Fast Fourrier Transform (FFT) operations.

This application contains a producer, several workers, several consumers and a remote entity as shown in Figure 1.

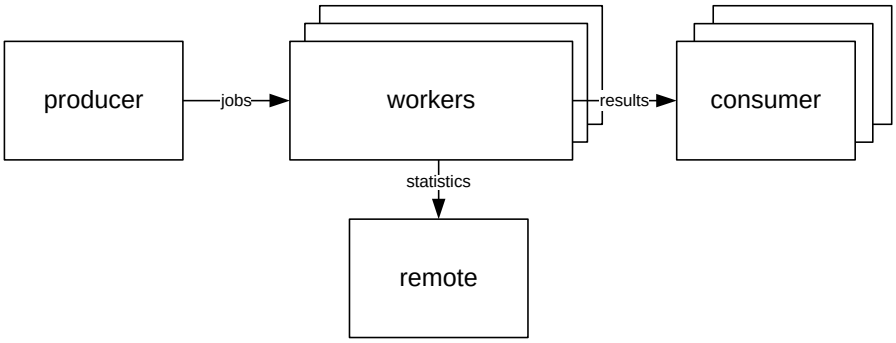


Figure 1: High level block diagram

- The producer generates one sequence of MY_JOB_NUM jobs equally distributed over MY_FLOW_NUM flows. One job triggers one complete FFT operation. The input data of the FFT are stored in the job. The size of the FFT is a power of two randomly distributed between MY_FFT_SIZE_MIN and MY_FFT_SIZE_MAX. The size of the FFT is also stored in the job. There is no dependency between jobs. Once all jobs have been generated, the producer becomes a worker.

- The workers are inactive during the complete duration of jobs generation. The workers consume the jobs, compute the FFTs and forward the results to the consumers. They generate statistics that are stored in a shared memory. Once activated, the workers are asynchronous. A worker is allowed to consume jobs from any flow.

- The consumers are inactive during the complete duration of FFTs computation. They consume and process the FFT results. In the example, this processing checks that the FFT results are correct. Once activated, the consumers are asynchronous.

- The remote entity is inactive during the complete duration of FFT results checking operation. It processes the job statistics from the shared memory.

5 Installing OpenEM

This section of the document details the procedure to install the OpenEM when it is delivered as a standalone package. Check the OpenEM release notes document to get the versions of the other TI tools that are required to be installed. This installation procedure is not required when the OpenEM is delivered through a MCSDK package.

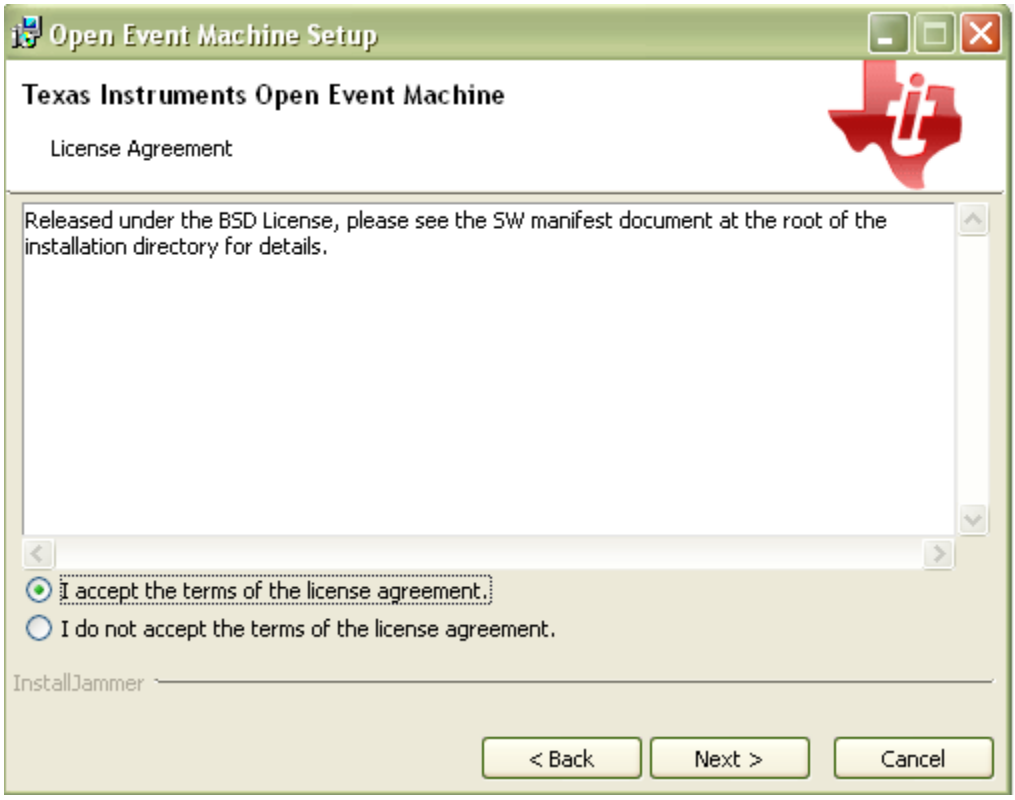
On Windows platforms, OpenEM is delivered with the "openem_w_x_y_z_k1_SetupWin32.exe" installer file for Keystone I devices and "openem_w_x_y_z_k2_SetupWin32.exe" installer file for Keystone II devices.

On Linux platforms, OpenEM is delivered with the "openem_w_x_y_z_k1_Linux-x86_Install.bin" installer file for Keystone I devices and "openem_w_x_y_z_k2_Linux-x86_Install.bin" installer file for keystone II devices.

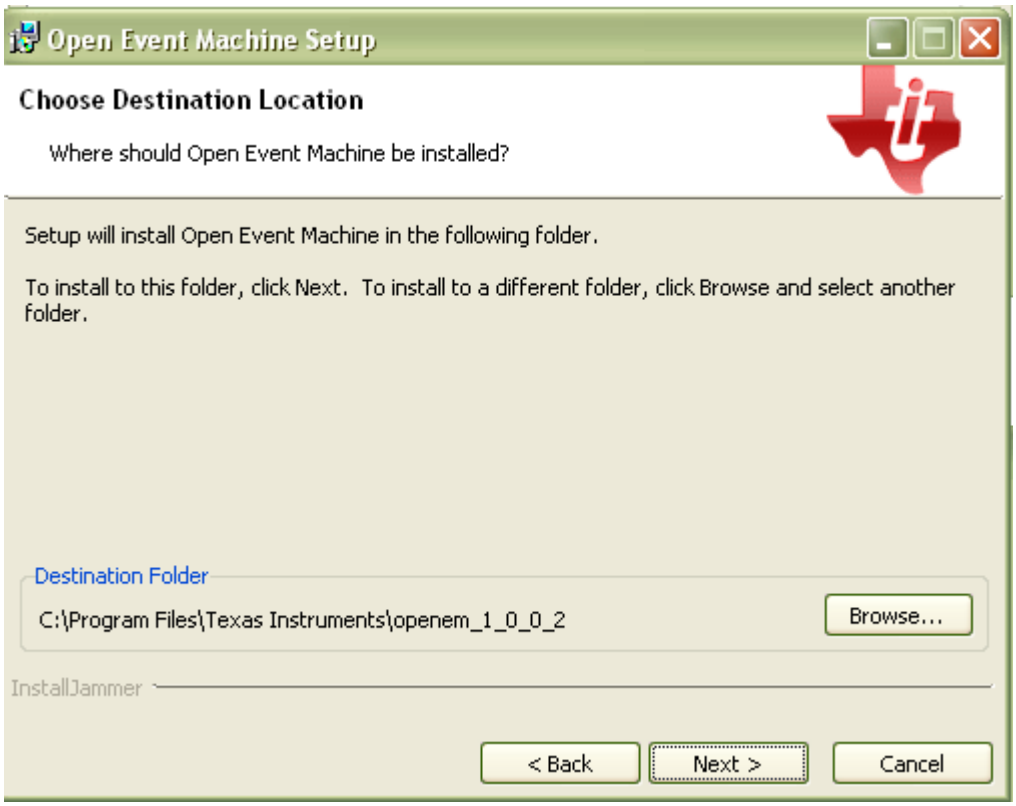
The following installation procedure applies to the OpenEM 1.0.0.2 on Keystone II devices.

When executing the installer on a Windows platform, the following InstallJammer Wizard starts:

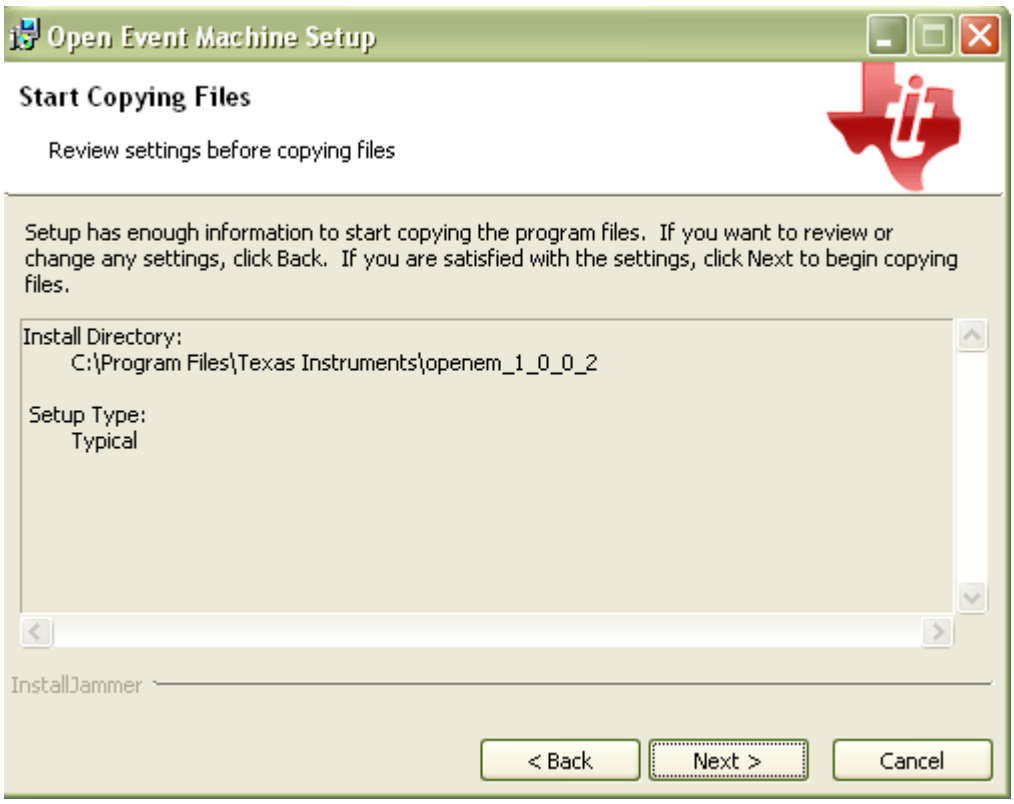
- It indicates the version of the OpenEM it is installing. Click on Next.



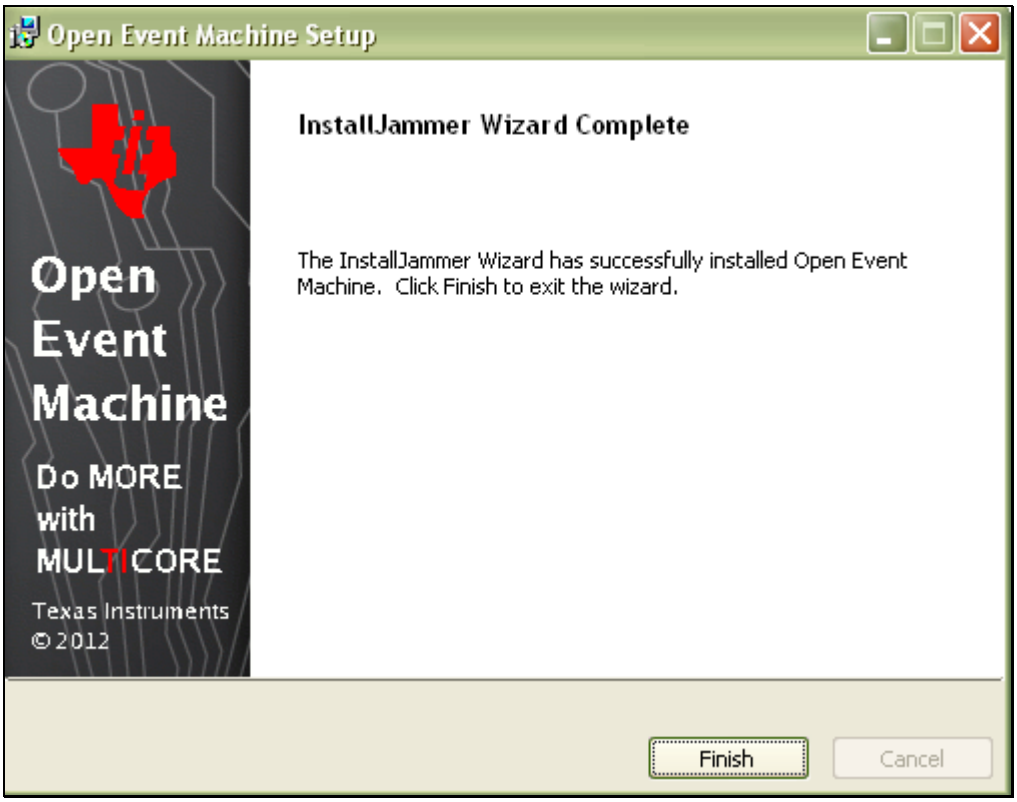
- Select “I accept the terms of the license agreement.” and click on Next.



- Select the destination folder for the OpenEM and click on Next.



- Check the summary of information and click on Next.



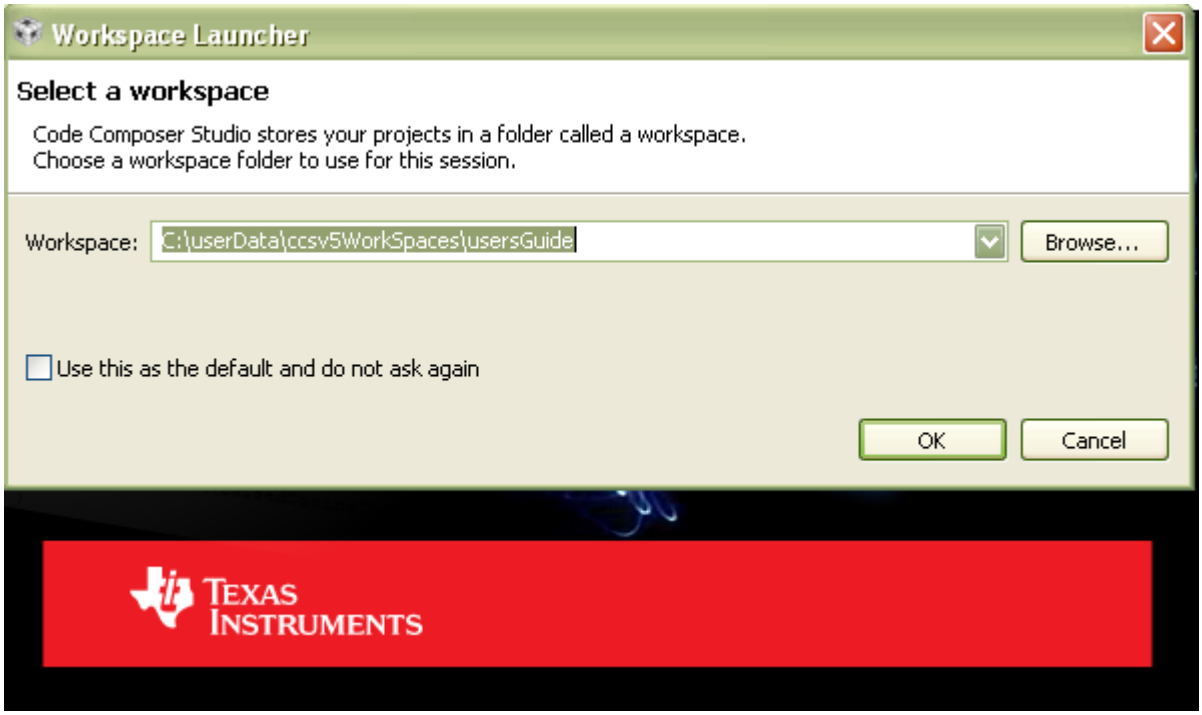
- The OpenEM is now installed, click on Finish.

6 OpenEM on DSP

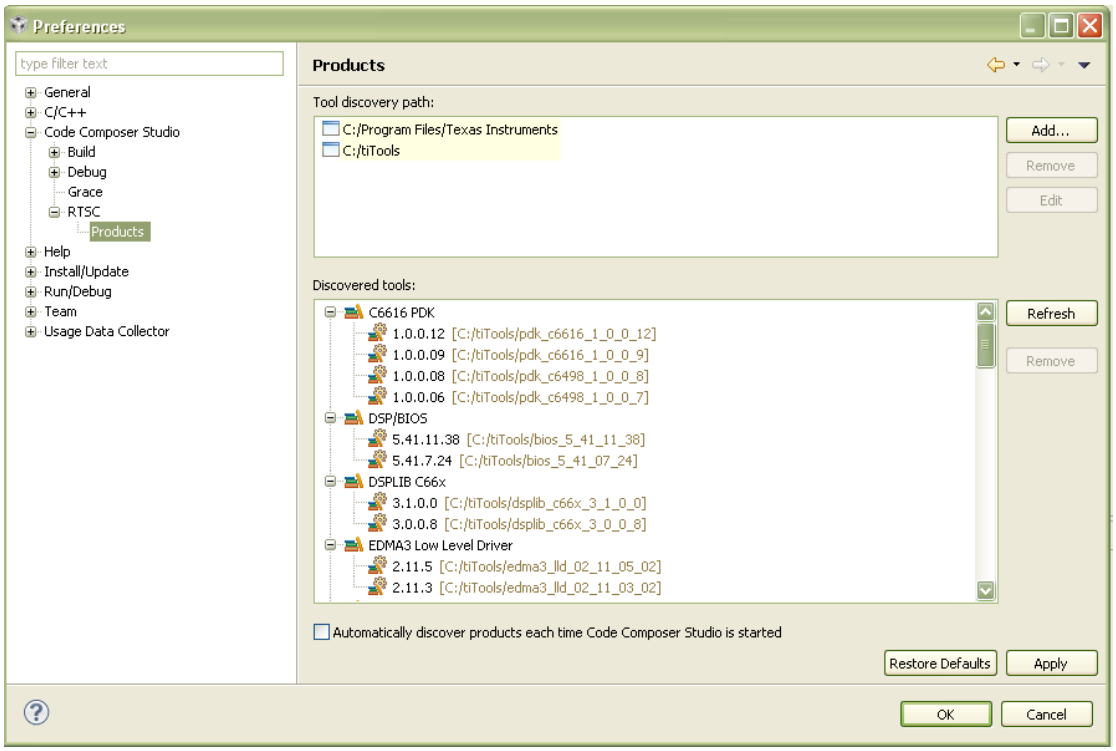
This section of the document details how to configure the examples on Code Composer Studio (CCS).

6.1 Installing OpenEM in CCS

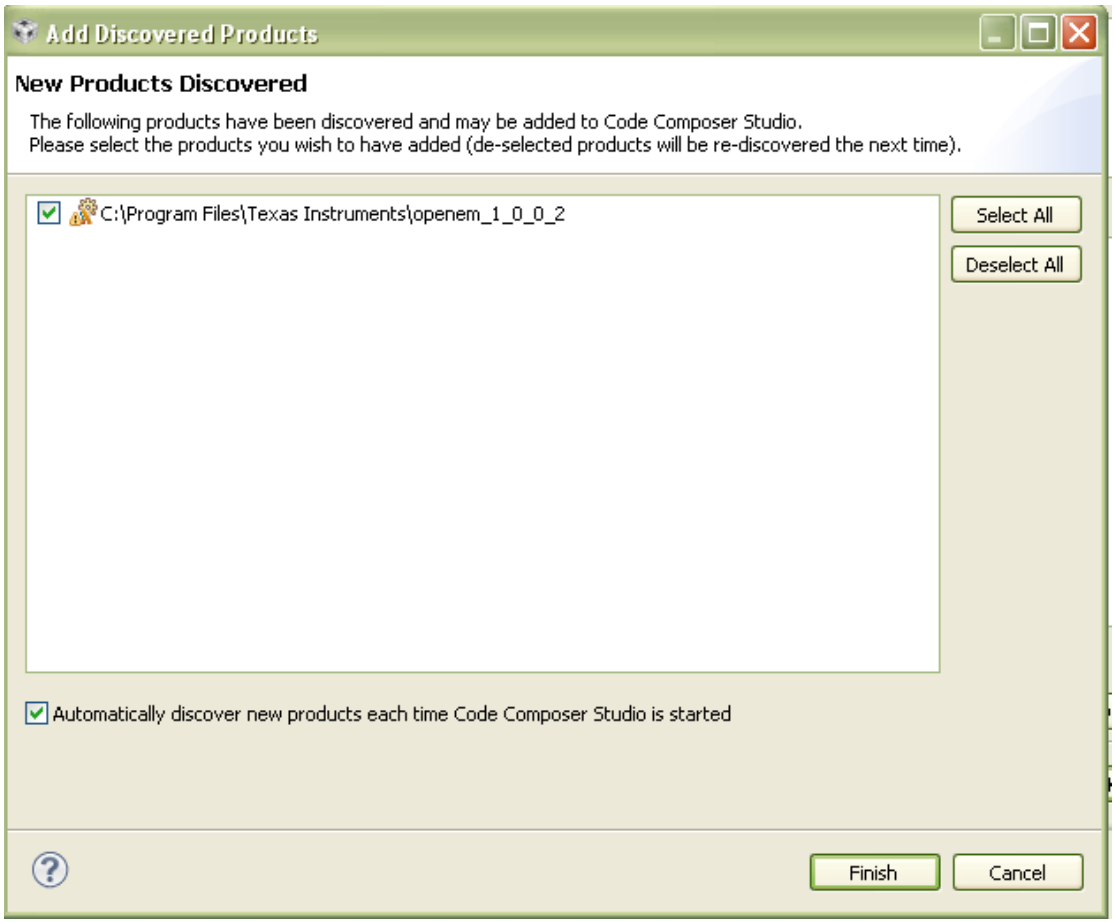
Start CCS. Select your workspace.



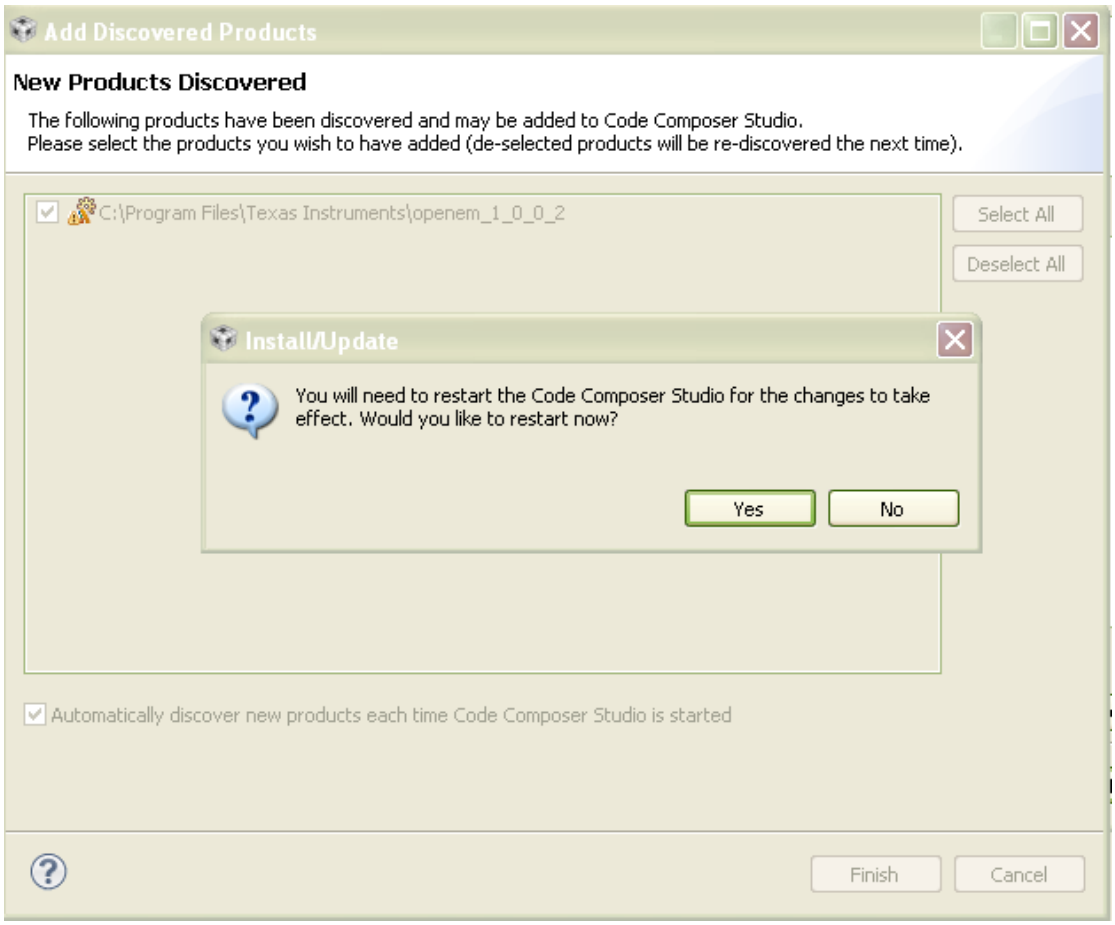
- If the automatic "new product detection" is not activated, go to window -> preferences, then Code Composer Studio -> RTSC -> Products. The following window shall appear. OpenEM is not listed in the Discovered tools.



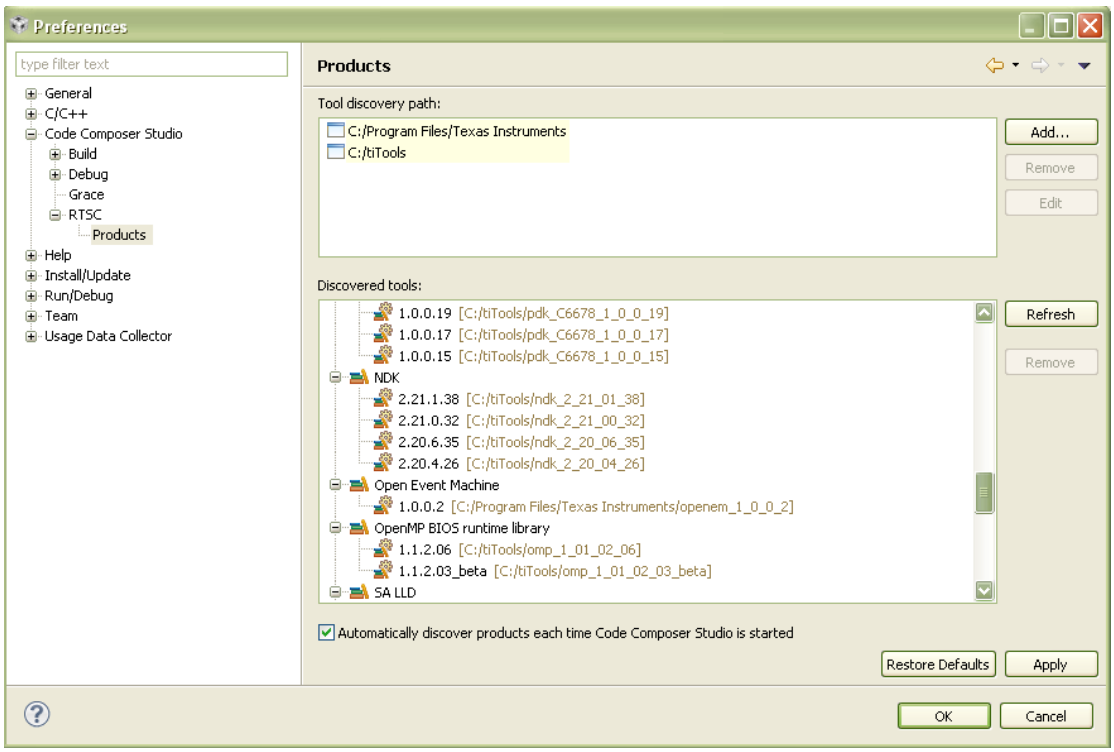
- Check the Tool discovery path is correctly set to the OpenEM package path or add it, click on Refresh.



- CCS detects the OpenEM as a new product. Click on Finish. If the automatic "new product detection" is activated, this window pops up automatically when the Tool discovery path is correctly set to the OpenEM package path.



- Click on Yes to restart CCS.



- CCS now recognizes the OpenEM as a valid RTSC package to be used in projects.

6.2 Managing examples in CCS

Following procedure details how to import, build and run an example project in CCS. Same procedure applies to all examples.

6.2.1.1 Import projects

To import the project in CCS:

- In CCS, click on the menu "File -> Import…", Select "Code Composer Studio -> Existing CCS/CCE Eclipse Projects" and click on Next.
- Click on the "Browse" button next to "Select search-directory", select the example directory under "C:\ti\openem_w_x_y_z\packages\ti\runtime\openem\dsp\examples". The project appears in the "Discovered projects" tab. Make sure it is selected and click on Finish.
- The project now appears in the "Project Explorer" tab of the "C/C++ Edit" perspective as a RTSC project and is ready to be configured and built.

6.2.1.2 Build Project

To configure the project for the build:

- To select the active build configuration, right click on the project name -> properties, then, click on Manage Configurations..., and select the build configuration (Debug, Release), click on Set Active and click OK.
- Ensure the correct versions of the PDK and OpenEM RTSC packages are selected by right clicking on project name -> properties -> General -> RTSC.
- Ensure the correct memory platform is selected by right clicking on project name -> properties -> General ->RTSC. For an unknown reason, it is possible the required memory platform does not appear in the drop down menu, in which case the memory platform name is of the form "ti.runtime.openem.dsp.examples.platforms.c6678_ex0".
- Ensure the symbolic constants in the file "my_event_machine.h" are correctly set and save the file. Those constants are part of the application and configure the OpenEM inline functions.
- Ensure the OpenEM is correctly configured.
    - Right click on the file Example.cfg -> Open With -> XGCONF, this will make the "Available Products" tab to appear in CCS.

- In this tab, click on the arrows next to Other Products->openem_x_y_z_w->ti->runtime->openem.
- Click on Settings, this will make the Example.cfg->Settings tab to appear.
- Set the correct values for the debugFlag and the targetFlag and save.

To proceed through the build of the project:

- Clean the project by right clicking on Example_0 -> Clean Project.
- Build the project by right clicking on Example_0 -> Build Project.

The build generates a binary file under Binaries and a build directory.

6.2.1.3 Run project

The run procedure of the project is the standard procedure, it requires selecting a ccxml file, connecting the target, loading the program and running the program.

6.3 Examples

The implementation of the OpenEM examples requires memory allocation, configuration and initialization of the OpenEM process, creation and activation of the OpenEM objects.

6.3.1 Abstraction layer

In order to hide to the user the complexity of initializing the QMSS, PKTDMA and OpenEM, a software abstraction layer is provided with the examples. The user configures a set of OpenEM parameters that allows the abstraction layer to initialize the QMSS, PKTDMA and OpenEM entities. The abstraction layer is part of each example code and is not part of the OpenEM library.

The abstraction layer provides three structures to be configured and three public APIs to be called by the user to fully manage the QMSS, PKTDMA and OpenEM entities.

- ti_em_pool_config2_t
- ti_em_pl_pool_config2_t
- ti_em_hw_config2_t
- ti_em_init_global2()
- ti_em_init_local2()

- ti_em_exit_global2()

The abstraction layer sets the value of the constant TI_EM_EVENT_NUM2 as the max number of events allocated in parallel from all application event pools (in Example_0, there are two application event pools, the public event pool and the exit event pool).

The abstraction layer sets the value of the constant Emti_LOCAL_EVENT_NUM as the number of events of the preload event pools.

The abstraction layer assumes the memory sections listed in 6.3.3.3 are mapped on the memory areas listed in 6.3.3.1.

The abstraction layer instantiates and manages the arrays of event descriptors used by the OpenEM for the job processing.

The abstraction layer does not instantiate the array of event buffers because there sizes are application specific.

6.3.1.1.1 ti_em_pool_config2_t

This structure contains the parameters required by the abstraction layer to configure a public event pool. Figure 2 shows the ti_em_pool_config2_t prototype.

```
typedef struct ti_em_pool_config2_t_
{
    uint32_t event_num;
    uint8_t *buffer_ptr;
    uint32_t buffer_size;
    uint32_t ps_word_num;
    uint32_t free_policy;
    ti_em_pool_config_t pool_config;
} ti_em_pool_config2_t;
```

Figure 2: ti_em_pool_config2_t

- event_num
  - Number of events in the event pool.

- buffer_ptr
  - Pointer to the array of event buffers attached to the event pool when the event pool buffer mode is set to

TI_EM_BUF_MODE_GLOBAL_TIGHT. It is set to NULL the event pool buffer mode is set to TI_EM_BUF_MODE_GLOBAL_LOOSE.

- buffer_size
  - Size of one event buffer in bytes. It is set to 0 when the buffer_ptr is set to NULL.

- ps_word_num
  - Number of protocol specific word. Refer to the QMSS and CPPI specifications for details.

- free_policy
  - Event free policy. Refer to the QMSS and CPPI specifications for details.

- pool_config
  - OpenEM event pool configuration. Refer to the OpenEM API specification for details.

6.3.1.1.2 ti_em_pl_pool_config2_t

This structure contains the parameters required by the abstraction layer to configure a preload event pool. Figure 3 shows the ti_em_pl_pool_config2_t prototype.

```
typedef struct ti_em_pl_pool_config2_t_
{
    uint32_t core_idx;
    uint32_t event_num;
    uint8_t *buffer_ptr;
    uint32_t buffer_size;
    uint32_t ps_word_num;
    uint32_t free_queue_idx;
} ti_em_pl_pool_config2_t;
```

Figure 3: ti_em_pl_pool_config2_t

- core_idx
  - Dispatcher core index.

- event_num
  - Number of events supported by the preload event pool.

- buffer_ptr
  - Pointer to the array of event buffers attached to the preload event pool.

- buffer_size

- Size of one event buffer in bytes.

- ps_word_num
  - Number of protocol specific word. Refer to the QMSS and CPPI specifications for details.

- free_queue_idx
  - QMSS general purpose hardware queue index allocated to the preload event pool. Free events are stored in the preload event pool free queue.

6.3.1.1.3 ti_em_hw_config2_t

This structure contains the parameters required by the abstraction layer to configure the OpenEM. Figure 3 shows the ti_em_hw_config2_t prototype.

```
typedef struct ti_em_hw_config2_t_
{
    uint32_t ap_private_free_queue_idx;
    uint32_t cd_private_free_queue_idx;
    uint32_t hw_queue_base_idx;
    uint32_t hw_sem_idx;
    uint32_t dma_queue_base_idx;
    uint32_t preload_size_a;
    uint32_t preload_size_b;
    uint32_t preload_size_c;
} ti_em_hw_config2_t;
```

Figure 4: ti_em_hw_config2_t

- ap_private_free_queue_idx
  - Index of the QMSS general purpose hardware queue containing the private events used for Atomic Processing. This queue shall contain at least 256 private events.

- cd_private_free_queue_idx
  - Index of the QMSS general purpose hardware queue containing the private events used for Command Processing. This queue shall contain at least 32 private events. This index can be the same as ap_private_free_queue_idx.

- hw_queue_base_idx
  - Base index for the TI_EM_HW_QUEUE_NUM contiguous QMSS general purpose hardware queues required by the OpenEM.

- hw_sem_idx
  - Index of the OpenEM hardware semaphore.

- dma_queue_base_idx
  - Base index for the TI_EM_DMA_TX_QUEUE_NUM contiguous infrastructure PKTDMA hardware transmit queues.

- preload_size_a
  - Preload size A.

- preload_size_b
  - Preload size B.

- preload_size_c
  - Preload size C.

6.3.1.1.4 ti_em_init_global2()

Purpose of ti_em_init_global2() is to configure the QMSS and PKTDMA hardware IP blocks and to initialize the QMSS, PKTDMA and OpenEM global shared variables.

It shall be called one and only one time by the master core.

ti_em_init_global2() gathers all parameters the user shall configure to initialize the OpenEM.

Figure 5 shows the prototype of ti_em_init_global2().

```
em_status_t ti_em_init_global2 (
    uint32_t region_num,
    Qmss_MemRegInfo * region_config_tbl,
    uint32_t pdsp_num,
    Qmss_PdspCfg * pdsp_config_tbl,
    uint32_t pool_num,
    ti_em_pool_config2_t * pool_config2_tbl,
    uint32_t core_num,
    ti_em_pl_pool_config2_t * pl_pool_config2_tbl,
    ti_em_hw_config2_t hw_config2
);
```

Figure 5: ti_em_init_global2()

ti_em_init_global2() requires the following parameters to be provided:

- region_num

    - Number of QMSS memory regions the application uses for its processing. It does not include the QMSS memory regions required by the OpenEM.

- region_config_tbl
    - Table of parameters to configure the memory regions the application uses for its processing. Qmss_MemRegInfo is detailed in QMSS LLD API specifications. If region_num is set to 0, this parameter is set to NULL.

- pdsp_num
    - Number of PDSPs the application uses for its processing. It does not include the PDSPs required by the OpenEM.

- pdsp_config_tbl
    - Table of parameters to configure the PDSPs the application uses for its processing. Qmss_PdspCfg is detailed in QMSS LLD API specification. If pdsp_num is set to 0, this parameter is set to NULL.

- pool_num
    - Number of application event pools. It does not include the event pools required for the preloading of the events.

- pool_config2_tbl
    - Table of parameters to configure the application event pools.

- core_num
    - Number of cores involved in the dispatching of events.

- pl_pool_config2_tbl
    - Table of parameters to configure the preloading event pools.

- hw_config2
    - Set of parameters to configure the OpenEM.

6.3.1.1.5 ti_em_init_local2()

Purpose of ti_em_init_local2() is to initialize the QMSS, PKTDMA and OpenEM local variables on each core dispatching events. It shall be called one and only one time per core after the ti_em_init_global2() is complete on the master core. It does not require input parameters.

Figure 6 shows the ti_em_init_local2() prototype.

```
em_status_t ti_em_init_local2 (
    void
);
```

Figure 6: ti_em_init_local2()

6.3.1.1.6 ti_em_exit_global2()

Purpose of ti_em_exit_global2() is to close the QMSS, CPPI and OpenEM software objects and to release the hardware resources used by the OpenEM.

It shall be called one and only one time by the master core.

Figure 5 shows the prototype of ti_em_exit_global2().

```
em_status_t ti_em_exit_global2 (
    void
);
```

Figure 7: ti_em_exit_global2()

6.3.2 File organization

Most of the example files have been named to identify their content. Groups of files are dedicated to the initialization abstraction layer (ref. 6.3.3), to the OpenEM initialization and to the application.

- The files dedicated to the abstraction layer are prefixed with “ti_em_”. The user should not modify these files.

- The files dedicated to the OpenEM initialization are prefixed with “my_em_”. Variables required by the OpenEM, but that are dependent on the application (e.g. “my_em_svProcEventBufMem[]”) are listed in these files. Also, all symbolic constants required by the OpenEM are listed in these files.

- Most of the files dedicated to the Application are prefixed by “my_”. Files related to the FFT processing are not prefixed.

- The other files

- my_event_machine.h is required and contains symbolic constants for configuring the OpenEM. It can be modified by the application.
- qmss_device.c and cppi_device.c are required to build the Example_0 for Keystone I devices. For Keystone II devices, original PDK files are included from ti_em_init.c. They contain the device specific configuration for the QMSS and CPPI Low Level Drivers.
- osal.c contains an Operating System Adaptation layer which is used by the QMSS and CPPI low level driver.
- em_pdk_hal.c contains the implementation of the OpenEM functions pointers to abstract the QMSS and CPPI low level drivers.
- Example_*x*.cfg contains the memory configuration for example "*x*".
    - It defines and maps the example memory sections on the memory areas defined in the memory platforms.
    - It lists and configures the RTSC modules the project is dependent on.
        - xdc.useModule('ti.csl.Settings');
        - xdc.useModule('ti.drv.cppi.Settings');
        - xdc.useModule('ti.drv.qmss.Settings');
        - xdc.useModule('ti.runtime.openem.Settings');

6.3.3 Memory management

The OpenEM examples requires local versus shared and cached versus non cached memories to be allocated. For that purpose, memory areas are created and memory sections are mapped to these memory areas.

These memory sections are either:

- predefined by the QMSS, PKTDMA and OpenEM libraries,
- predefined by the abstraction layer,
- defined by the application.

6.3.3.1 Memory areas

A set of custom memory areas is created to manage "internal" versus "external" memories and cached versus non cached memories.

These memory areas are defined in the memory platforms delivered along with the OpenEM package in directory

$(OpenEMRootDirectory)\packages\ti\runtime\openem\dsp\examples\platforms. For Example_*x*, memory platform are post fixed "_ex*x*".

Memory areas are:

- L1PSRAM
  - Used as cache memory, no length defined.
- L1DSRAM
  - Used as cache memory, no length defined.
- L2SRAM
  - memory : UMC RAM
  - cache : enabled
- MSMCSRAM
  - memory : MSMC RAM
  - cache : enabled
- MSMCSRAM_NC
  - memory : MSMC RAM
  - cache : disabled
- DDR3
  - memory : DDR3
  - cache : enabled
- DDR3_NC
  - memory : DDR3
  - cache : disabled
- PDSP1D
  - memory : PDSP1 RAM
  - cache : NA
- PDSP2D
  - memory : PDSP2 RAM
  - cache : NA
- PDSP3D
  - Keystone II only
  - memory : PDSP3 RAM
  - cache : NA
- PDSP4D
  - Keystone II only
  - memory : PDSP4 RAM
  - cache : NA
- PDSP5D
  - Keystone II only
  - memory : PDSP5 RAM
  - cache : NA
- PDSP6D
  - Keystone II only
  - memory : PDSP6 RAM
  - cache : NA
- PDSP7D
  - Keystone II only

    - memory : PDSP7 RAM
    - cache : NA
- PDSP8D
    - Keystone II only
    - memory : PDSP8 RAM
    - cache : NA
- PDSPSH
    - Keystone II only
    - memory : PDSP shared RAM
    - cache : NA

When using PDSP memory areas with DSP BIOS, it is mandatory to disable caching and prefetching on MAR52 (0X34000000) for Keystone I devices and MAR35 (0x23000000) for Keystone II devices. Not doing so will result in unpredictable behavior of the QMSS when pushing and popping descriptors to QMSS hardware queues. In the examples, it is done at runtime when initializing memories.

6.3.3.2 Libraries memory sections

QMSS, PKTDMA and OpenEM libraries require their predefined memory sections to be mapped to the custom memory areas listed above. These memory sections contain dedicated shared variables and local variables that are declared, defined and used within the scope of these libraries.

- QMSS
    - .qmss
        - memory: MSMCSRAM_NC

- PKTDMA
    - .cppi
        - memory: MSMCSRAM_NC

- OpenEM
    - . tiEmGlobalFast
        - memory: MSMCSRAM_NC
        - size: my_em_getGlobalSizeFast()

    - . tiEmGlobalSlow
        - memory : DDR3_NC
        - size: my_em_getGlobalSizeSlow()

    - . tiEmLocal
        - memory : L2SRAM
        - size: my_em_getLobalSize()

6.3.3.3 Abstraction layer memory sections

The abstraction layer requires its predefined memory sections to be mapped to the custom memory areas listed above.

- .tiEmSvPrivateEventDscMem1
  - memory: PDSP1D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem2
  - memory: PDSP2D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem3
  - memory: PDSP3D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem4
  - memory: PDSP4D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem5
  - memory: PDSP5D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem6
  - memory: PDSP6D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem7
  - memory: PDSP7D
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPrivateEventDscMem8
  - memory: PDSP8D

- size: TI_EM_PDSP_GLOBAL_DATA_SIZE
- alignment: CACHE_L2_LINESIZE

- . tiEmSvPrivateEventDscMem
  - Keystone II devices only
  - memory: PDSPSH
  - size: TI_EM_PDSP_GLOBAL_DATA_SIZE
  - alignment: CACHE_L2_LINESIZE
- .tiEmSvPublicEventDscMem
  - memory: MSMCSRAM_NC / DDR3_NC
  - size: (1 + TI_EM_EVENT_NUM2) * MY_EM_PUBLIC_EVENT_DSC_SIZE
  - The first descriptor of any memory region storing public and preload events is reserved to the OpenEM usage.
  - alignment: CACHE_L2_LINESIZE
- .tiEmGvLocalEventDscMem
  - memory: L2SRAM
  - size: (1 + Emti_LOCAL_EVENT_NUM) * MY_EM_PUBLIC_EVENT_DSC_SIZE
  - The first descriptor of any memory region storing public and preload events is reserved to the OpenEM usage.
  - alignment: CACHE_L2_LINESIZE

6.3.3.4 Application memory sections

On top of its own memory sections dedicated to its own processing, the application shall also map other additional memory sections dedicated to the OpenEM.

These memory sections contain the arrays of event buffers for both the public events and the preload events. The size of the event buffers being application specific, these sections cannot be transferred neither in the context of the OpenEM library, neither in the context of the abstraction layer.

- .my_em_sv_ProcEventBufMem
  - Location for the array of public event buffers (my_em_svProcEventBufMem[])
  - memory: MSMCRAM / MSMCRAM _NC / DDR3 / DDR3_NC
  - size: MY_EM_EVENT_NUM * MY_EM_EVENT_BUF_SIZE
  - alignment: CACHE_L2_LINESIZE
- .my_em_gvLocalEventBufMem

- Location for the array of preload event buffers (my_em_gvPlEventBufMem[])
- memory: L2SRAM
- size: MY_EM_PL_EVENT_NUM * MY_EM_PL_EVENT_BUF_SIZE
- alignment: CACHE_L2_LINESIZE

6.3.4 Example_0

The Example_0 implements the application described in the introduction of the document.

Figure 8 illustrates the Example_0 this implementation.

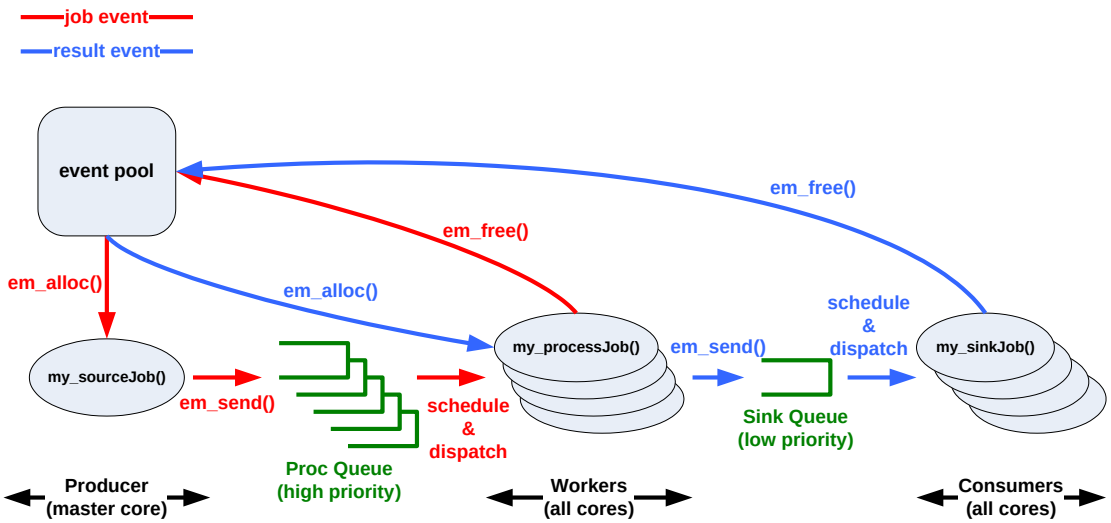


Figure 8 : OpenEM implementation

- The producer is a DSP core (master). It calls a function “my_sourceJob()” that performs the complete jobs generation. For each job, “my_sourceJobs()” calls “em_alloc()” to allocate an event from an event pool, fills the event buffers of this event with control information and input data, calls “em_send()” to send the event to one of the “Proc Queue” event queues (EQ). There is one “Proc Queue” EQ per job flow.

- The workers are DSP cores. They consume the events coming from the “Proc Queue” EQs. Upon reception of an event, a worker calls a function “my_processJob()”. “my_processJob()” is the receive function of the execution object (EO) linked to the “Proc Queue” EQs. “my_processJob()” processes the FFT, calls “em_alloc()” to allocate a new event from the event pool to stores the FFT results and calls “em_send()” to send this event to a single “Sink Queue” EQ. It calls “em_free()” to free the received event.

- The consumers are DSP cores. They consume the events coming from the “Sink Queue” EQ. Upon reception of an event, a consumer calls a function “my_sinkJob()”. “my_sinkJob()” is the receive function of the EO linked to the “Sink Queue” EQ. “my_sinkJob()” checks the correctness of the results. It calls “em_free()” to free the received event.

- The remote entity is implemented by the master core. Statistics are processed after the FFT results have been checked. It does not involve the OpenEM.

- Preloading mechanism is not illustrated on the figure.

On top of this implementation, an exit procedure based on the OpenEM has been implemented. It is initiated by the master core after the statistics have been computed. Figure 9 illustrates the exit procedure.

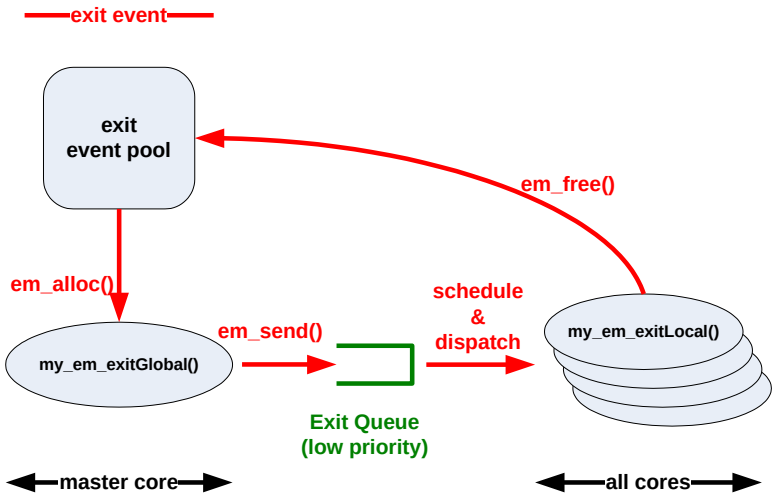


Figure 9 : Exit procedure

- The master core calls the function “my_em_exitGlobal()”. “my_em_exitGlobal()” calls “em_alloc()” to allocate (MY_EM_CORE_NUM-1) events from an exit event pool and calls “em_send()” to send the events to a single “Exit Queue” EQ.

- The dispatcher cores consume the events coming from the “Exit Queue” EQ. Upon reception of an event, a core calls a function “my_em_exitLocal()”. “my_em_exitLocal()” is the receive function of the EO linked to the “Exit Queue” EQ. “my_em_exitLocal()” exits the program.

- Once all dispatchers have exited the program, the master core also exits the program.

This implementation requires the following OpenEM objects to be created by the application:

- Events

- (2 * MY_JOB_NUM) events to store the input and output data of the MY_JOB_NUM FFTs. They are called public events.
- (MY_EM_CORE_NUM – 1) events for the exit procedure. They are called exit events.

- Event pools
  - 1 public event pool to store the public events.
  - 1 exit event pool to store the exit events.

- EOs
  - 1 “Proc EO” EO associated to the “my_processJob()” receive function.
  - 1 “Sink EO” EO associated to the “my_sinkJob()” receive function.
  - 1 “Exit EO” EO associated to the “my_em_exitLocal()” receive function.

- EQs
  - MY_EM_PROC_QUEUE_NUM “Proc Queues” linked to the “Proc EO” EO.
  - 1 “Sink Queue” linked to the “Sink_EO” EO.
  - 1 “Exit Queue” linked to the “Exit_EO” EO.

The Example_0 covers the following OpenEM features:

- Parallel or atomic event queues
  - The “Proc Queue” event queues are created as parallel or atomic queues according to the value of the symbolic constant MY_EM_PROC_QUEUE_TYPE defined in my_em_init.h.

- Event preloading
  - The overall mechanism for preloading is always configured in Example_0.
  - Preloading is enabled or disabled for the public events according to the value of the symbolic constant MY_EM_PROC_EVENT_TYPE defined in my_em_init.h.

- Multi-threaded scheduler
  - The multi-threaded scheduler is configurable on Keystone II devices only. A single scheduler is used for Keystone I devices.

- The number of scheduler threads is configured according to the value of the symbolic constants MY_EM_SCHEDULER_THREAD_NUM defined in my_em_init.h.

Figure 10, Figure 11 and Figure 12 show the main procedure of Example_0.

```
int
main (
    void
)
{
    /*
     * initialization
     */
    if (my_init () < 0)
        return -1;

    /*
     * on master core
     */
    if (DNUM == MY_EM_INIT_CORE_IDX)
    {
        /*
         * initialize fft shared variables
         */
        if (my_initFft () < 0)
            return -1;
        /*
         * initialize jobs
         */
        if (my_initJobs () < 0)
            return -1;
        /*
         * allocates events, initialize event buffers,
         * * send events
         */
        if (my_sourceJobs () < 0)
            return -1;

        /*
         * synchronisation barrier
         */
        my_waitAtCoreBarrier ();
```

Figure 10: main() – 1/3

```
    /*
     * request first event from scheduler
     */
    ti_em_preschedule ();

    /*
     * call event dispatcher
     */
    while (my_svRunningJobNum > 0)
        ti_em_dispatch_once ();

    /*
     * compute job statictics
     */
    if (my_statsJobs () < 0)
    {
        printf ("Debug(Core %d): my_statsJobs failed.\n", DNUM);
        printf ("Debug(Core %d): \
            <CompletionTag> Example #0 failed.\n", DNUM);
        return -1;
    }

    /*
     * terminate program
     */
    if (my_exit () < 0)
    {
        printf ("Debug(Core %d): my_exit failed.\n", DNUM);
        printf ("Debug(Core %d): \
            <CompletionTag> Example #0 failed.\n", DNUM);
        return -1;
    }

    printf ("Debug(Core %d): \
        <CompletionTag> Example #0 passed.\n", DNUM);
    printf ("\n==========\n");

}
```

Figure 11: main() - 2/3

```
    /*
     * on dispatcher core
     */
    else
    {
        /*
         * synchronisation barrier
         */
        my_waitAtCoreBarrier ();

        /*
         * request first event from scheduler
         */
        ti_em_preschedule ();

        /*
         * call event dispatcher
         */
        while (1)
            ti_em_dispatch_once ();
    }

    return 0;
}
```

Figure 12: main() - 3/3

- All cores enter the initialization procedure.
- The master core initializes the application and sends the public events.
- All cores dispatch the public events.
- The master core computes the statistics.
- The master core enters the exit procedure.

6.3.4.1 Initialization procedure

The OpenEM initialization is divided in one global initialization and one local initialization per core. It is simplified by the usage of the software abstraction layer that handles most of the QMSS, PKTDMA and OpenEM initialization steps.

The application is responsible for instantiating the arrays of event buffers. my_em_svProcEventBufMem[] and my_em_gvPlEventBufMem[] are instantiated in the my_em_init.c file.

```
/* array of application event buffers */
#pragma DATA_ALIGN   (my_em_svProcEventBufMem, CACHE_L2_LINESIZE)
#pragma DATA_SECTION (my_em_svProcEventBufMem, ".my_em_svProcEventBufMem");
uint8_t my_em_svProcEventBufMem[MY_EM_EVENT_NUM * MY_EM_EVENT_BUF_SIZE];

/* array of preload event buffers */
#pragma DATA_ALIGN   (my_em_gvPlEventBufMem, CACHE_L2_LINESIZE)
#pragma DATA_SECTION (my_em_gvPlEventBufMem, ".my_em_gvLocalEventBufMem");
uint8_t my_em_gvPlEventBufMem[MY_EM_PL_EVENT_NUM * MY_EM_PL_EVENT_BUF_SIZE];
```

Figure 13: event buffer instantiations

In Example_0, the OpenEM initialization is located in my_init() and implemented by two function calls my_em_initGlobal() and my_em_initLocal().

Figure 14 and Figure 15 show the implementation of my_init().

```
int
my_init (
    void
)
{
    /*
     * initialize memory per core
     */
    if (my_initMemLocal () < 0)
        return -1;

    /*
     * synchronisation barrier
     */
    my_waitAtCoreBarrier ();

    /*
     * on master core
     */
    if (DNUM == MY_EM_INIT_CORE_IDX)
    {
        /*
         * initialize memory translation for QMSS
         */
        if (my_initMemGlobal () < 0)
            return -1;

        /*
         * OpenEM global initialization
         */
        if (my_em_initGlobal () < 0)
            return -1;

        /*
         * application timer global initialization
         */
        if (my_initClockGlobal () < 0)
            return -1;
    }

    my_waitAtCoreBarrier ();
```

Figure 14: my_init() 1/2

```
    /*
     * OpenEM local initialization
     */
    if (my_em_initLocal () < 0)
        return -1;

    /*
     * application timer local initialization
     */
    if (my_initClockLocal () < 0)
        return -1;

    my_waitAtCoreBarrier ();

    /*
     * on master core
     */
    if (DNUM == MY_EM_INIT_CORE_IDX)
    {
        /*
         * OpenEM objects creation
         */
        if (my_em_initQueues () < 0)
            return -1;
    }

    return 0;
}
```

Figure 15: my_init() 2/2

- my_em_initGlobal() is called on the master core.
- my_em_initLocal() is called on all cores after the global initialization is complete.

6.3.4.1.1 OpenEM global initialization

In Example_0, the implementation of the OpenEM global initialization is performed by my_em_initGlobal().

Figure 16 shows the implementation of my_em_initGlobal().

```
int
my_em_initGlobal (
    void
)
{
    ti_em_pool_config2_t lvPoolConfig2Tbl[MY_EM_POOL_NUM];
    ti_em_pl_pool_config2_t lvPlPoolConfig2Tbl[MY_EM_CORE_NUM];
    ti_em_hw_config2_t lvHwConfig2;

    /*
     * set pools initialization parameters
     */
    my_em_initPoolConfig (lvPoolConfig2Tbl);

    /*
     * set preload pools initialization parameters
     */
    my_em_initPlPoolConfig (lvPlPoolConfig2Tbl);

    /*
     * set OpenEM initialization parameters
     */
    my_em_initHwConfig (&lvHwConfig2);

    /*
     * call OpenEM global initialization
     */
    if (ti_em_init_global2(
            0, NULL,
            0, NULL,
            MY_EM_POOL_NUM, lvPoolConfig2Tbl,
            MY_EM_CORE_NUM, lvPlPoolConfig2Tbl,
            lvHwConfig2) != EM_OK)
        return -1;

    return 0;
}
```

Figure 16: my_em_initGlobal()

- The three configuration structures of the abstraction layer are instantiated.

- my_em_initPoolConfig() configures the table of parameters for the public event pools.

- my_em_initPlPoolConfig() configures the table of parameters for the preload event pools.

- my_em_initHwConfig() configure the table of parameters of the OpenEM.

- ti_em_init_global2() is a call to the abstraction layer API that initializes the QMSS, PKTDMA and OpenEM using the table of parameters previously configured.
    - No other QMSS memory regions are used by the application.
    - No other PDSPs are used by the application.
    - MY_EM_POOL_NUM application event pools are used by the application.
    - MY_EM_CORE_NUM cores are configured for event preloading.

6.3.4.1.1.1 my_em_initPoolConfig

This function configures the parameters for the two public event pools that the application uses for the job processing.

Figure 17 shows the implementation of my_em_initPoolConfig().

```
void
my_em_initPoolConfig (
    ti_em_pool_config2_t * poolConfig2Tbl
)
{
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].pool_config.free_queue_idx =
            MY_EM_FREE_QUEUE_BASE_IDX + MY_EM_EXIT_POOL_IDX;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].pool_config.buf_mode =
            TI_EM_BUF_MODE_GLOBAL_TIGHT;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].pool_config.dsc_wsize =
            (size_t)(MY_EM_PUBLIC_EVENT_DSC_SIZE);

    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].event_num = MY_EM_EXIT_EVENT_NUM;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].free_policy = 0;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].buffer_ptr = NULL;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].buffer_size = 0;
    poolConfig2Tbl[MY_EM_EXIT_POOL_IDX].ps_word_num = 0;

    poolConfig2Tbl[MY_EM_POOL_IDX].pool_config.free_queue_idx =
            MY_EM_FREE_QUEUE_BASE_IDX + MY_EM_POOL_IDX;
    poolConfig2Tbl[MY_EM_POOL_IDX].pool_config.buf_mode = MY_EM_BUF_MODE;
    poolConfig2Tbl[MY_EM_POOL_IDX].pool_config.dsc_wsize =
            (size_t)(MY_EM_PUBLIC_EVENT_DSC_SIZE);

    poolConfig2Tbl[MY_EM_POOL_IDX].event_num = MY_EM_EVENT_NUM;
    poolConfig2Tbl[MY_EM_POOL_IDX].free_policy = 0;
    poolConfig2Tbl[MY_EM_POOL_IDX].buffer_ptr = my_em_svProcEventBufMem;
    poolConfig2Tbl[MY_EM_POOL_IDX].buffer_size = MY_EM_EVENT_BUF_SIZE;
    poolConfig2Tbl[MY_EM_POOL_IDX].ps_word_num = 0;
}
```

Figure 17: my_em_initPoolConfig()

Though the "buf_mode" parameter is set to TI_EM_BUF_MODE_GLOBAL_TIGHT for the "Exit" event pool, the buffer pointer for this pool is set to NULL. Therefore, events allocated from this pool act as tokens.

6.3.4.1.1.2 my_em_initPlPoolConfig

Figure 18 shows the implementation of my_em_initPlPoolConfig().

```
void
my_em_initPlPoolConfig (
    ti_em_pl_pool_config2_t * poolConfig2Tbl
)
{
    int i;

    for (i = 0; i < MY_EM_CORE_NUM; i++)
    {
        poolConfig2Tbl[i].core_idx = i;
        poolConfig2Tbl[i].event_num = MY_EM_PL_EVENT_NUM;
        poolConfig2Tbl[i].buffer_ptr = (uint8_t*)my_em_makeAddressGlobal
                (i, (uint32_t)my_em_gvPlEventBufMem);
        poolConfig2Tbl[i].buffer_size = MY_EM_PL_EVENT_BUF_SIZE;
        poolConfig2Tbl[i].ps_word_num = 0;
        poolConfig2Tbl[i].free_queue_idx = MY_EM_PL_FREE_QUEUE_BASE_IDX + i;
    }
}
```

Figure 18: my_em_initPlPoolConfig()

6.3.4.1.1.3 my_em_initHwConfig

Figure 19 shows the implementation of my_em_initHwConfig().

```
void
my_em_initHwConfig (
    ti_em_hw_config2_t * hwConfig2Ptr
)
{
    hwConfig2Ptr->ap_private_free_queue_idx = MY_EM_AP_PRIVATE_FREE_QUEUE_IDX;
    hwConfig2Ptr->cd_private_free_queue_idx = MY_EM_CD_PRIVATE_FREE_QUEUE_IDX;
    hwConfig2Ptr->hw_queue_base_idx = MY_EM_HW_QUEUE_BASE_IDX;
    hwConfig2Ptr->hw_sem_idx = MY_EM_HW_SEM_IDX;
    hwConfig2Ptr->dma_queue_base_idx = MY_EM_DMA_QUEUE_BASE_IDX;
    hwConfig2Ptr->preload_size_a = MY_EM_PRELOAD_SIZE_A;
    hwConfig2Ptr->preload_size_b = MY_EM_PRELOAD_SIZE_B;
    hwConfig2Ptr->preload_size_c = MY_EM_PRELOAD_SIZE_C;
    return;
}
```

Figure 19: my_em_initHwConfig()

6.3.4.1.2 OpenEM local initialization

The local initialization is implemented by my_em_initLocal(). It calls ti_em_init_local2() from the initialization abstraction layer.

Figure 20 shows the implementation of my_em_initLocal().

```
int
my_em_initLocal (
    void
)
{
    /*
     * call OpenEM local initialization
     */
    if (ti_em_init_local2 () != EM_OK)
        return -1;
    return 0;
}
```

Figure 20: my_em_initLocal()

6.3.4.2 OpenEM objects creation

For the purpose of Example_0, several EOs, several EQs and one queue group shall be created; there is no need for event groups. They are created by the master core at the beginning of the scenario by calling my_initQueues(). Figure 21, Figure 22, Figure 23 and Figure 24 show the implementation of my_initQueues()

These OpenEM objects are “semi static” objects in the sense that once created, they cannot be deleted. Deleting them at runtime would require a synchronization procedure between cores that is not implemented in the current version of the OpenEM. Nevertheless, these objects can be created at any time during the life of the application code.

```
int
my_em_initQueues (
    void
)
{
    uint32_t i;
    em_queue_group_t exit_queue_group;

    /*
     * "Proc EO" creation
     */
    my_em_svProcEoHdl = em_eo_create(
            "Proc EO",
            my_em_eoStartDefault, NULL,
            my_em_eoStopDefault, NULL,
            my_processJob, NULL);
    if (my_em_svProcEoHdl == EM_EO_UNDEF)
        return -1;

    /*
     * loop over all Proc Queue
     */
    for (i = 0; i < MY_EM_PROC_QUEUE_NUM; i++)
    {
        /*
         * "Proc Queue" create
         */
        my_em_svProcQueueHdlTbl[i] = em_queue_create(
            "Proc Queue",
            MY_EM_PROC_QUEUE_TYPE,
            EM_QUEUE_PRIO_HIGH,
            EM_QUEUE_GROUP_DEFAULT);
        if (my_em_svProcQueueHdlTbl[i] == EM_QUEUE_UNDEF)
            return -1;
```

Figure 21: my_em_initQueues() - 1/4

```
    /*
     * add "Proc Queue" to "Proc EO"
     */
    if (em_eo_add_queue(
        my_em_svProcEoHdl,
        my_em_svProcQueueHdlTbl[i]) != EM_OK)
        return -1;
}

/*
 * "Proc EO" start
 */
if (em_eo_start(
        my_em_svProcEoHdl,
        NULL,
        0,
        NULL) != EM_OK) return -1;

/*
 * "Sink EO" creation
 */
my_em_svSinkEoHdl = em_eo_create(
        "Sink EO",
        my_em_eoStartDefault, NULL,
        my_em_eoStopDefault, NULL,
        my_sinkJob, NULL);
if (my_em_svSinkEoHdl == EM_EO_UNDEF)
    return -1;

/*
 * "Sink Queue" create
 */
my_em_svSinkQueueHdl = em_queue_create(
        "Sink Queue",
        EM_QUEUE_TYPE_PARALLEL,
        EM_QUEUE_PRIO_LOW,
        EM_QUEUE_GROUP_DEFAULT);
if (my_em_svSinkQueueHdl == EM_QUEUE_UNDEF)
    return -1;
```

Figure 22: my_em_initQueues() - 2/4

```
/*
 * add "Sink Queue" to "Sink EO"
 */
if (em_eo_add_queue(
        my_em_svSinkEoHdl,
        my_em_svSinkQueueHdl) != EM_OK)
    return -1;

/*
 * "Sink EO" start
 */
if (em_eo_start(
        my_em_svSinkEoHdl,
        NULL,
        0,
        NULL) != EM_OK) return -1;

/*
 * "Exit EO" creation
 */
my_em_svExitEoHdl = em_eo_create(
        "Exit EO",
        my_em_eoStartDefault, NULL,
        my_em_eoStopDefault, NULL,
        my_em_exitLocal, NULL);
if (my_em_svExitEoHdl == EM_EO_UNDEF)
    return -1;
/*
 * "Exit_Queue_Group" queue group creation
 */
{
    em_core_mask_t lvMask;
    em_core_mask_zero (&lvMask);

    for (i = 1; i < MY_EM_CORE_NUM; i++)
        em_core_mask_set (i, &lvMask);
    exit_queue_group = em_queue_group_create (
        "Exit_Queue_Group",
        (const em_core_mask_t *) &lvMask,
        0,
        NULL);
}
```

Figure 23: my_em_initQueues() - 3/4

```
    /*
     * "Exit Queue" create
     */
    my_em_svExitQueueHdl = em_queue_create (
            "Exit Queue",
            EM_QUEUE_TYPE_ATOMIC,
            EM_QUEUE_PRIO_LOWEST,
            exit_queue_group);
    if (my_em_svExitQueueHdl == EM_QUEUE_UNDEF)
        return -1;

    /*
     * add "Exit Queue" to "Exit EO"
     */
    if (em_eo_add_queue (
            my_em_svExitEoHdl,
            my_em_svExitQueueHdl) != EM_OK)
        return -1;

    /*
     * "Exit EO" start
     */
    if (em_eo_start (
            my_em_svExitEoHdl,
            NULL,
            0,
            NULL) != EM_OK)
        return -1;

    return 0;
}
```

Figure 24: my_em_initQueues() - 4/4

6.3.4.2.1 EOs

EOs are created using em_eo_create(). EQs are added to the EOs using em_eo_add_queue(), finally, EOs are started using em_eo_start().

When created and started, the EO is associated to a receive function that is executed by one of the dispatcher cores after an event is pushed into one of the added EQs.

All EOs of Example_0 share the same global and local start and stop functions, but they differ with their receive functions.

- "Proc EO"
  - This EO is linked to the my_processJob() receive function. This receive function executes in the following order:
    - A prefetch request to the scheduler. When getting this prefetch request, the scheduler prefetches the next event for this dispatcher core and eventually preloads this event.
    - If the preload is enabled, the receive function calls the ti_em_claim_local() API to get access to the data preloaded in the local event buffer.
    - The allocation of a new event for storing the FFT results.
    - The FFT processing.
    - The dump of statistics data in shared memory.
    - The free of the received event.
    - The send of the new event into the “Sink Queue”.
  - Figure 25, Figure 26, Figure 27 and Figure 28 show the implementation of my_processJob().

```
void
my_processJob (
    void *eoCtxtPtr,
    em_event_t eventHdl,
    em_event_type_t eventType,
    em_queue_t queueHdl,
    void *queueCtxtPtr
)
{
    uint8_t *lvJobBufPtr;
    my_JobDsc *lvJobDscPtr;
    int16_t *lvDataInputTbl;
    int16_t *lvDataOutputTbl;
    volatile em_event_t lvOutputEventHdl;
    uint32_t lvJobIdx;
    uint32_t lvFftSize;
    uint32_t lvDataSize;
    uint32_t lvSuccessFlag;
    uint32_t lvStartTime;
    uint32_t lvStopTime;

    /*
     * prefetch request
     */
    ti_em_preschedule ();

    /*
     * access event buffer of received event.
     * Could be a local buffer or the buffer of the
     * global event.
     */
    if (ti_em_get_type_preload (eventType) !=
        TI_EM_EVENT_TYPE_PRELOAD_OFF)
        lvJobBufPtr = (uint8_t *) ti_em_claim_local ();
    else
        lvJobBufPtr = (uint8_t *) em_event_pointer (
            (em_event_t) eventHdl);

    lvJobDscPtr = (my_JobDsc *) lvJobBufPtr;
    lvJobBufPtr = lvJobBufPtr + MY_JOB_HDR_SIZE;
    lvDataInputTbl = (int16_t *) lvJobBufPtr;
```

Figure 25: my_processJob() - 1/4

```
/*
 * retrieve ctrl information from received
 * event buffer
 */
lvJobIdx = lvJobDscPtr->jobIdx;
lvFftSize = lvJobDscPtr->fftSize;
lvDataSize = lvJobDscPtr->dataSize;
lvSuccessFlag = lvJobDscPtr->successFlag;

/*
 * allocate a new event for storing FFT results
 */
lvOutputEventHdl = EM_EVENT_UNDEF;
do
lvOutputEventHdl = em_alloc(
        lvDataSize,
        TI_EM_EVENT_TYPE_PRELOAD_OFF,
        MY_EM_POOL_IDX);
while (lvOutputEventHdl == EM_EVENT_UNDEF);

/*
 * access event buffer of new event
 */
lvJobBufPtr = (uint8_t *) em_event_pointer
    (lvOutputEventHdl);
lvJobDscPtr = (my_JobDsc *) lvJobBufPtr;
lvJobBufPtr = lvJobBufPtr + MY_JOB_HDR_SIZE;
lvDataOutputTbl = (int16_t *) lvJobBufPtr;

/*
 * fill new event buffer with ctrl information
 */
lvJobDscPtr->jobIdx = lvJobIdx;
lvJobDscPtr->fftSize = lvFftSize;
lvJobDscPtr->dataSize = lvDataSize;
lvJobDscPtr->successFlag = lvSuccessFlag;

/*
 * get start time for statistics
 */
lvStartTime = my_readClockGlobal ();
```

Figure 26: my_processJob() - 2/4

```
/*
 * process the FFT
 */
switch (lvFftSize)
{
    case MY_FFT_SIZE_MIN:
    DSP_fft16x16( my_svTwiddleTbl0, lvFftSize,
                  lvDataInputTbl, lvDataOutputTbl);
    break;
case 2*MY_FFT_SIZE_MIN:
    DSP_fft16x16( my_svTwiddleTbl1, lvFftSize,
                  lvDataInputTbl, lvDataOutputTbl);
    break;
case 4*MY_FFT_SIZE_MIN:
    DSP_fft16x16( my_svTwiddleTbl2, lvFftSize,
                  lvDataInputTbl, lvDataOutputTbl);
    break;
case 8*MY_FFT_SIZE_MIN:
    DSP_fft16x16( my_svTwiddleTbl3, lvFftSize,
                  lvDataInputTbl, lvDataOutputTbl);
       break;
}

/*
 * get stop time for statistics
 */
lvStopTime = my_readClockGlobal ();

/*
 * store information for statistics
 */
my_svJobDscTbl[lvJobIdx].coreIdx = DNUM;
my_svJobDscTbl[lvJobIdx].startTime = lvStartTime;
my_svJobDscTbl[lvJobIdx].stopTime = lvStopTime;
```

Figure 27: my_processJob() - 3/4

```
    /*
     * free received event
     */
    em_free (eventHdl);

    /*
     * send new event
     */
    em_send (lvOutputEventHdl, my_em_svSinkQueueHdl);
}
```

Figure 28: my_processJob() - 4/4

- "Sink EO"
  - This execution object is linked to the my_sinkJob() receive function. This receive function executes in the following order:
    - A prefetch request to the scheduler.
    - The check of the FFT results.
    - The free of the receive event.
    - Decrements a shared “running jobs” counter. Access to this shared variable is protected using a semaphore.
  - Figure 1 and Figure 30 show the implementation of my_sinkJob().

```
void
my_sinkJob (
    void *eoCtxtPtr,
    em_event_t eventHdl,
    em_event_type_t eventType,
    em_queue_t queueHdl,
    void *queueCtxtPtr
)
{
    uint8_t *lvJobBufPtr;
    my_JobDsc *lvJobDscPtr;
    int16_t *lvDataTbl;
    uint32_t lvJobIdx;
    uint32_t lvFftSize;
    uint32_t i;

    /*
     * prefetch request
     */
    ti_em_preschedule ();

    /*
     * access event buffer of received event
     */
    lvJobBufPtr = (uint8_t *) em_event_pointer (eventHdl);
    lvJobDscPtr = (my_JobDsc *) lvJobBufPtr;
    lvJobBufPtr = lvJobBufPtr + MY_JOB_HDR_SIZE;
    lvDataTbl = (int16_t *) lvJobBufPtr;
```

Figure 29: my_sinkJob() – 1/2

```
    /*
     * retrieve ctrl information from received
     * event buffer
     */
    lvJobIdx = lvJobDscPtr->jobIdx;
    lvFftSize = lvJobDscPtr->fftSize;

    my_svJobDscTbl[lvJobIdx].successFlag = 1;

    /*
     * check results
     */
    for (i = 0; i < lvFftSize; i++)
    {
        if (i == (lvFftSize >> 2))
        {
            if (lvDataTbl[2 * i] == 0)
                my_svJobDscTbl[lvJobIdx].successFlag = 0;
            if (lvDataTbl[2 * i + 1] != 0)
                my_svJobDscTbl[lvJobIdx].successFlag = 0;
        }
        else
        {
            if (lvDataTbl[2 * i] != 0)
                my_svJobDscTbl[lvJobIdx].successFlag = 0;
            if (lvDataTbl[2 * i + 1] != 0)
                my_svJobDscTbl[lvJobIdx].successFlag = 0;
        }
    }

    /*
     * free received event
     */
    em_free (eventHdl);

    /*
     * decrement shared job counter
     */
    my_lock ();
    my_svRunningJobNum--;
    my_unlock ();
}
```

Figure 30: my_sinkJob() – 2/2

- "Exit EO"

- This execution object is linked to the my_em_exitLocal() receive function. This receive function exists the dispatcher core.
- Figure 31 shows the implementation of my_em_exitLocal().

```
void
my_em_exitLocal (
    void *eoCtxtPtr,
    em_event_t eventHdl,
    em_event_type_t eventType,
    em_queue_t queueHdl,
    void *queueCtxtPtr
)
{
    em_free (eventHdl);
    my_em_svExitStatus++;
    em_atomic_processing_end ();
    abort ();
}
```

Figure 31: my_em_exitLocal()

6.3.4.2.2 EQs

EQs created by the application are logical queues, not QMSS hardware queues. Therefore, the application can create several thousands of EQs. Ultimately, the EQs are mapped to the QMSS hardware queues according to their priorities and core masks.

EQs are created using em_queue_create(). Then they must be added to an EO before they can be used.

An EQ belongs to a queue group to which is attached a core mask. This core mask allows selecting which dispatcher cores will be eligible for getting events from this EQ. Queue groups are dynamically created, excepted for the default queue group that exists by default. The default queue group is “all cores eligible” to the EQ. An EQ can be parallel or atomic. An EQ has a priority.

EQs created for the purpose of Example_0 are:

- "Proc Queue"
    - MY_EM_PROC_QUEUE_NUM “Proc Queue” EQs are created.
    - These EQs are of type MY_EM_PROC_QUEUE_TYPE that is configured as parallel or atomic at compile time.

- They are queues with the high priority.
- They belong to the default queue group enabling all dispatcher cores to dispatch events from these EQs.
- They are added to the "Proc EO" EO, meaning an event pushed into one of these EQs will trigger a call to my_receiveJob().

- "Sink Queue"
  - One “Sink Queue” EQ is created.
  - This EQ is a parallel EQ.
  - It is a queue with the low priority.
  - It belongs to the default queue group.
  - It is added to the "Sink EO" EO, meaning an event pushed into this EQ will trigger a call to my_sinkJob().

- "Exit Queue"
  - One “Exit Queue” EQ is created.
  - This EQ is an atomic EQ.
  - It is a queue with the lowest priority.
  - It belongs to the “Exit_Queue_Group” queue group.
  - It is added to the “Exit EO" EO, meaning an event pushed into this EQ will trigger a call to my_em_exitLocal().

6.3.4.2.3 OpenEM activation

Activating the OpenEM means allocating and sending events to the EQs and configuring cores as dispatchers to process these events.

In Example_0, the allocation and send of the first set of events, corresponding to the jobs creation of Figure 1, is performed in my_sourceJobs() by the master core.

Figure 32, Figure 33 and Figure 34 show the implementation of my_sourceJobs().

```
int
my_sourceJobs (
    void
)
{
    uint32_t i, j;

    /*
     * initialize job counter
     */
    my_svRunningJobNum = MY_JOB_NUM;

    /*
     * loop over all jobs to create
     */
    for (i = 0; i < MY_JOB_NUM; i++)
    {
        volatile em_event_t lvEventHdl;
        uint8_t *lvJobBufPtr;
        my_JobDsc *lvJobDscPtr;
        int16_t *lvDataTbl;
        uint32_t lvFlowIdx;
        uint32_t lvFftSize;
        uint32_t lvDataSize;
        uint32_t lvSuccessFlag;

        lvFlowIdx = my_svJobDscTbl[i].flowIdx;
        lvFftSize = my_svJobDscTbl[i].fftSize;
        lvDataSize = my_svJobDscTbl[i].dataSize;
        lvSuccessFlag = my_svJobDscTbl[i].successFlag;

        /*
         * allocate event
         */
        lvEventHdl = EM_EVENT_UNDEF;
        do
        lvEventHdl = em_alloc(lvDataSize,
                              MY_EM_PROC_EVENT_TYPE,
                              MY_EM_POOL_IDX);
        while (lvEventHdl == EM_EVENT_UNDEF);
```

Figure 32: my_sourceJobs() – 1/3

```
if (lvEventHdl == NULL)
    return -1;

/*
 * acess event buffer
 */
lvJobBufPtr = (uint8_t *) em_event_pointer (lvEventHdl);

/*
 * set ctrl information
 */
lvJobDscPtr = (my_JobDsc *) lvJobBufPtr;
lvJobDscPtr->jobIdx = i;
lvJobDscPtr->flowIdx = lvFlowIdx;
lvJobDscPtr->fftSize = lvFftSize;
lvJobDscPtr->dataSize = lvDataSize;
lvJobDscPtr->successFlag = lvSuccessFlag;
```

Figure 33: my_sourceJobs() - 2/3

```
            /*
             * set FFT input data
             */
            lvJobBufPtr = lvJobBufPtr + MY_JOB_HDR_SIZE;
            lvDataTbl = (int16_t *) lvJobBufPtr;
            for (j = 0; j < lvFftSize; j++)
            {
                switch (j & 0x3)
                {
                    case 0:
                        *lvDataTbl++ = 1;
                        *lvDataTbl++ = 0;
                        break;
                    case 1:
                        *lvDataTbl++ = 0;
                        *lvDataTbl++ = 1;
                        break;
                    case 2:
                        *lvDataTbl++ = (-1);
                        *lvDataTbl++ = 0;
                        break;
                    case 3:
                    default:
                        *lvDataTbl++ = 0;
                        *lvDataTbl++ = (-1);
                        break;
                }
            }

            /*
             * send event to EQs
             */
            em_send(lvEventHdl,
                    my_em_svProcQueueHdlTbl[lvFlowIdx]);
        }

        return 0;
    }
```

Figure 34: my_sourceJobs() - 3/3

Once my_sourceJobs returns, the master core becomes a dispatcher core. As for the other dispatcher cores, it first calls ti_em_preschedule() to request an event from the OpenEM scheduler, then it calls ti_em_dispatch_once() to dispatch and process an event scheduled by the OpenEM scheduler.

The activation of the dispatcher cores is performed in main().

6.3.4.3 Summary

Listed below are the symbolic constants, variables and functions to be implemented to run Example_0 with OpenEM.

6.3.4.3.1 Symbolic constants

- MY_EM_CORE_NUM (TI_EM_CORE_NUM)
  - Number of cores involved with the OpenEM, TI_EM_CORE_NUM is an OpenEM symbolic constant.
  - Set to 4 when using C6670

- MY_EM_INIT_CORE_IDX (0)
  - Master core Index
  - Hard coded to core 0 for Example_0 as for the exit procedure, the queue group does not include core 0.

- MY_EM_PROC_QUEUE_NUM (32)
  - Number of high priority parallel “Proc Queue” EQs that need to be created.

- MY_EM_PROC_QUEUE_TYPE (EM_QUEUE_TYPE_PARALLEL)
  - Type of the MY_EM_PROC_QUEUE_NUM Proc Queue” EQs.

- MY_EM_PROC_EVENT_TYPE (TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_C)
  - Type of the events that are allocated in my_sourceJobs() when creating the jobs. It indicates event preloading is enabled.

- MY_EM_PUBLIC_EVENT_DSC_SIZE (64)
  - Size of the event descriptors in bytes. This value is used to dimension the array of descriptors.

- MY_EM_POOL_NUM (2)
  - Number of event pools required by the application. One is the public event pool containing all the events involved in the job processing. The other is the exit event pool to exit the application.

- MY_EM_POOL_IDX (0)
    - Index of the public event pool containing the public event involved in the job processing.

- MY_EM_EVENT_NUM (2048)
    - Number of public events in the public event pool.

- MY_EM_EVENT_BUF_SIZE (32*1024+CACHE_L2_LINESIZE)
    - Public event buffer size of the public events of the public event pool.

- MY_EM_BUF_MODE (TI_EM_BUF_MODE_GLOBAL_TIGHT)
    - Public event pool buffer mode.

- MY_EM_COH_MODE (TI_EM_COH_MODE_ON)
    - Public event pool coherency mode.

- MY_EM_EXIT_POOL_IDX (1)
    - Index of the exit event pool.

- MY_EM_EXIT_EVENT_NUM (MY_EM_CORE_NUM)
    - Number of exit events in the exit event pool.

- MY_EM_PRELOAD_SIZE_A (256)
    - Number of bytes for the preload size A

- MY_EM_PRELOAD_SIZE_B (2*1024)
    - Number of bytes for the preload size B

- MY_EM_PRELOAD_SIZE_C (64*1024)
    - Number of bytes for the preload size C

- MY_EM_ PRELOAD_EVENT_NUM (2)
    - Number of preload events in the preload event pool. This value is hardcoded and cannot be changed by the user.

- MY_EM_ PRELOAD_EVENT_BUF_SIZE (MY_EM_PRELOAD_SIZE_C)
    - Preload event buffer size of the preload events of the preload event pool.

- MY_EM_AP_PRIVATE_FREE_QUEUE_IDX (1022)
    - Index of the QMSS general purpose hardware queue containing the private events used for Atomic Processing. This queue shall contain at least 256 private events.

- MY_EM_CD_PRIVATE_FREE_QUEUE_IDX (1023)
    - Index of the QMSS general purpose hardware queue containing the private events used for Command Processing. This queue shall contain at least 32 private events. This index can be the same as MY_EM_AP_PRIVATE_FREE_QUEUE_IDX.

- MY_EM_HW_QUEUE_BASE_IDX (1024)
    - QMSS general purpose queue base index for OpenEM internal processing. It shall be aligned on multiple of 128.

- MY_EM_FREE_QUEUE_BASE_IDX (2048)
    - QMSS general purpose queue base index for the application event pools. These queues store the free events before they are allocated by the application.

- MY_EM_ PRELOAD_FREE_QUEUE_BASE_IDX (MY_EM_FREE_QUEUE_BASE_IDX + MY_EM_POOL_NUM)
    - QMSS general purpose queue base index for the preload event pools.

- MY_EM_DMA_QUEUE_BASE_IDX (0)
    - Relative QMSS transmit queues base index for preloading.

- MY_EM_HW_SEM_IDX (3)
    - OpenEM hardware semaphore.

- TI_EM_EVENT_NUM2 (MY_EM_EXIT_EVENT_NUM + MY_EM_EXIT_EVENT_NUM)
    - Total number of application events the application can allocate in parallel from the application event pools. This constant is required by the abstraction layer.

6.3.4.3.2 Variables and arrays

The application is responsible for instantiating the arrays of event buffers:

- my_em_svProcEventBufMem[MY_EM_EVENT_NUM * MY_EM_EVENT_BUF_SIZE]
  - Array of application event buffers

- my_em_gvPlEventBufMem[MY_EM_PL_EVENT_NUM * MY_EM_PL_EVENT_BUF_SIZE]
  - Array of preload event buffers

6.3.4.3.3 Functions

- em_status_t ti_em_init_global2()
  - Global initialization of the OpenEM.

- em_status_t ti_em_init_local2(void)
  - Local initialization of the OpenEM.

- em_status_t ti_em_exit_global2(void)
  - Global exit of the OpenEM.

- my_em_initQueues()
  - Create all OpenEM objects involved in the application processing

- my_processJobs(), my_sinkJobs(), my_em_exitLocal()
  - Application receive functions associated to the EOs created by my_em_initQueues()

- my_sourceJobs()
  - Allocates and sends public events to the OpenEM.

6.3.4.4 Outputs

Complete logs are provided below for several configurations.

- Each core returns
  - The total number of jobs it has processed.
  - The number of jobs per flow.
  - The Min/Mean/Max number of cycles to perform the job processing.
  - The Min/Mean/Max number of cycles consumed by the dispatcher.

- The OpenEM handles the event buffer cache coherency.

*6.3.4.4.1 Parallel queues – preload size (64*1024) – 1 scheduler thread*

```
This test-case uses the following configuration:
  - my_device_idx        : 0
  - my_process_idx       : 0
  - nb scheduler pdsp    : 1
  - thread #0 on pdsp    : 0
  - AP private_free_queue: 1022
  - CD private_free_queue: 1023
  - hw_queue_base_idx    : 1024
  - dma_queue_base_idx   : 0
  - pool_num             : 2
Init Done

==========
data check OK!

Statistics for core 0:
Number of jobs : 123
Number of jobs per flow: 3 4 6 7 2 4 2 5 2 4 3 5 2 3 2 4 2 4 5 3 6 5 3 4 3
3 4 3 7 5 4 4
Min/mean/max processing cycles : 5471/32323/78992
Min/mean/max overhead cycles : 2283/8378/37857


Statistics for core 1:
Number of jobs : 130
Number of jobs per flow: 4 4 4 7 5 5 3 4 4 4 2 4 3 2 7 3 5 2 5 4 6 2 3 7 6
3 6 4 3 4 2 3
Min/mean/max processing cycles : 5430/30542/78520
Min/mean/max overhead cycles : 2263/8119/40714


Statistics for core 2:
Number of jobs : 129
Number of jobs per flow: 4 4 2 2 2 2 8 4 5 2 2 3 5 9 4 4 4 6 2 2 4 4 7 3 4
4 5 6 5 3 4 4
Min/mean/max processing cycles : 5411/30199/78736
Min/mean/max overhead cycles : 2283/8645/38105


Statistics for core 3:
Number of jobs : 136
```

```
Number of jobs per flow: 5 6 2 2 4 5 4 8 3 4 3 3 5 3 5 2 4 5 6 5 7 3 4 4 4
4 3 8 4 1 5 5
Min/mean/max processing cycles : 5437/28737/77862
Min/mean/max overhead cycles : 2263/8639/32373


Statistics for core 4:
Number of jobs : 135
Number of jobs per flow: 3 2 6 4 7 4 5 2 8 5 8 5 5 4 3 5 3 2 6 5 1 6 4 2 3
3 3 2 4 6 4 5
Min/mean/max processing cycles : 5458/29398/76708
Min/mean/max overhead cycles : 2251/8221/31733


Statistics for core 5:
Number of jobs : 125
Number of jobs per flow: 3 5 7 3 5 3 5 4 4 4 3 5 5 4 5 4 6 3 4 3 3 3 3 3 3
3 5 5 2 3 4 3
Min/mean/max processing cycles : 5445/31226/78564
Min/mean/max overhead cycles : 2261/9250/37216


Statistics for core 6:
Number of jobs : 112
Number of jobs per flow: 4 5 3 4 3 5 1 2 4 4 3 4 3 2 3 4 4 4 2 4 2 6 3 7 5
4 2 2 4 2 5 2
Min/mean/max processing cycles : 5589/35195/78324
Min/mean/max overhead cycles : 2315/10184/36483


Statistics for core 7:
Number of jobs : 134
Number of jobs per flow: 6 2 2 3 4 4 4 3 2 5 8 3 4 5 3 6 4 6 2 6 3 3 5 2 4
8 4 2 3 8 4 6
Min/mean/max processing cycles : 5367/29065/79182
Min/mean/max overhead cycles : 2261/8541/30645


Statistics for all cores
Total number of jobs: 1024
Total number of jobs per flow: 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
Min/mean/max processing cycles : 5367/30721/79182
Min/mean/max overhead cycles : 2251/8718/40714

Start/stop/run cycles: 61441377/83040250/21598873
Debug(Core 0): <CompletionTag> Example #0 passed.

==========
```

*6.3.4.4.2 Atomic queues – preload size (64*1024) – 1 scheduler thread*

```
This test-case uses the following configuration:
  - my_device_idx        : 0
  - my_process_idx       : 0
  - nb scheduler pdsp    : 1
```

```
  - thread #0 on pdsp    : 0
  - AP private_free_queue: 1022
  - CD private_free_queue: 1023
  - hw_queue_base_idx    : 1024
  - dma_queue_base_idx   : 0
  - pool_num             : 2
Init Done

==========
data check OK!

Statistics for core 0:
Number of jobs : 128
Number of jobs per flow: 0 0 0 0 0 0 32 0 0 0 0 32 0 0 0 0 0 32 0 0 0 0 0 0
0 32 0 0 0 0 0 0
Min/mean/max processing cycles : 5477/29779/75840
Min/mean/max overhead cycles : 2289/9614/33311


Statistics for core 1:
Number of jobs : 128
Number of jobs per flow: 32 0 0 0 0 0 0 0 0 0 0 0 32 0 0 0 0 0 0 0 0 32
0 0 0 0 0 32 0 0
Min/mean/max processing cycles : 5541/29537/76492
Min/mean/max overhead cycles : 2263/9404/98521


Statistics for core 2:
Number of jobs : 128
Number of jobs per flow: 0 0 32 0 0 0 0 0 32 0 0 0 0 0 0 0 32 0 0 0 0 0 0 0
0 0 32 0 0 0 0 0
Min/mean/max processing cycles : 5365/28332/75520
Min/mean/max overhead cycles : 2267/9476/94877


Statistics for core 3:
Number of jobs : 128
Number of jobs per flow: 0 0 0 0 0 0 0 32 0 0 0 0 0 0 32 0 0 0 0 0 0 0 32 0
0 0 0 0 0 0 32 0
Min/mean/max processing cycles : 5473/31841/77854
Min/mean/max overhead cycles : 2255/9730/56523


Statistics for core 4:
Number of jobs : 128
Number of jobs per flow: 0 32 0 0 0 0 0 0 0 0 0 0 32 0 0 0 0 0 0 0 0 32 0 0
0 0 0 0 0 0 0 32
Min/mean/max processing cycles : 5413/33339/78114
Min/mean/max overhead cycles : 2269/10316/83855


Statistics for core 5:
Number of jobs : 128
Number of jobs per flow: 0 0 0 32 0 0 0 0 0 0 32 0 0 0 0 0 0 0 32 0 0 0 0 0
32 0 0 0 0 0 0 0
Min/mean/max processing cycles : 5443/30416/76502
Min/mean/max overhead cycles : 2300/9286/81657
```

```
Statistics for core 6:
Number of jobs : 128
Number of jobs per flow: 0 0 0 0 32 0 0 0 0 32 0 0 0 0 0 0 0 0 32 0 0 0 0
0 0 0 32 0 0 0 0
Min/mean/max processing cycles : 5365/29563/76584
Min/mean/max overhead cycles : 2273/9137/38183


Statistics for core 7:
Number of jobs : 128
Number of jobs per flow: 0 0 0 0 0 32 0 0 0 0 0 0 0 0 32 0 0 0 0 32 0 0 0
0 0 0 0 32 0 0 0
Min/mean/max processing cycles : 5393/32405/80924
Min/mean/max overhead cycles : 2253/9463/44483


Statistics for all cores
Total number of jobs: 1024
Total number of jobs per flow: 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
Min/mean/max processing cycles : 5365/30651/80924
Min/mean/max overhead cycles : 2253/9553/98521

Start/stop/run cycles: 61522296/79139376/17617080
Debug(Core 0): <CompletionTag> Example #0 passed.

==========
```

6.3.4.4.3 Parallel queues – preload off – 1 scheduler thread

```
This test-case uses the following configuration:
  - my_device_idx        : 0
  - my_process_idx       : 0
  - nb scheduler pdsp    : 1
  - thread #0 on pdsp    : 0
  - AP private_free_queue: 1022
  - CD private_free_queue: 1023
  - hw_queue_base_idx    : 1024
  - dma_queue_base_idx   : 0
  - pool_num             : 2
Init Done

==========
data check OK!

Statistics for core 0:
Number of jobs : 130
Number of jobs per flow: 5 3 1 7 4 3 4 2 4 3 9 7 2 4 4 4 4 4 1 4 3 5 3 4 3
5 6 3 4 5 5 5
Min/mean/max processing cycles : 6013/33879/88422
Min/mean/max overhead cycles : 2711/6123/12570


Statistics for core 1:
Number of jobs : 114
```

```
Number of jobs per flow: 6 5 4 4 2 3 5 3 5 1 4 4 3 4 1 3 8 3 3 4 4 2 4 7 2
5 2 1 4 2 3 3
Min/mean/max processing cycles : 6343/38430/89832
Min/mean/max overhead cycles : 2681/6679/13592


Statistics for core 2:
Number of jobs : 133
Number of jobs per flow: 3 3 7 4 6 3 4 5 4 2 6 5 4 4 5 5 4 3 7 6 2 7 5 6 5
3 2 2 3 2 3 3
Min/mean/max processing cycles : 6111/32538/85768
Min/mean/max overhead cycles : 2681/6046/15018


Statistics for core 3:
Number of jobs : 141
Number of jobs per flow: 3 5 3 4 3 6 4 3 5 5 2 3 3 3 6 3 6 5 4 5 2 4 3 4 10
7 3 5 4 8 5 5
Min/mean/max processing cycles : 6211/30396/84664
Min/mean/max overhead cycles : 2679/5839/16640


Statistics for core 4:
Number of jobs : 129
Number of jobs per flow: 3 7 3 3 4 5 7 6 3 4 2 1 6 5 5 5 2 4 4 3 3 3 3 2 4
2 10 6 4 1 5 4
Min/mean/max processing cycles : 6149/33033/86036
Min/mean/max overhead cycles : 2687/6545/16260


Statistics for core 5:
Number of jobs : 134
Number of jobs per flow: 5 3 7 4 3 6 3 5 2 4 2 6 4 5 3 6 1 5 1 5 6 4 4 2 3
4 5 9 4 5 4 4
Min/mean/max processing cycles : 6169/32112/85879
Min/mean/max overhead cycles : 2685/6514/17361


Statistics for core 6:
Number of jobs : 121
Number of jobs per flow: 3 2 5 3 5 2 4 5 2 6 4 1 6 4 5 2 6 2 6 3 7 5 4 3 3
2 3 3 5 4 3 3
Min/mean/max processing cycles : 6021/35673/92994
Min/mean/max overhead cycles : 2683/7120/16820


Statistics for core 7:
Number of jobs : 122
Number of jobs per flow: 4 4 2 3 5 4 1 3 7 7 3 5 4 3 3 4 1 6 6 2 5 2 6 4 2
4 1 3 4 5 4 5
Min/mean/max processing cycles : 6275/35197/90976
Min/mean/max overhead cycles : 2683/6875/19778


Statistics for all cores
Total number of jobs: 1024
Total number of jobs per flow: 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
Min/mean/max processing cycles : 6013/33763/92994
```

```
Min/mean/max overhead cycles : 2679/6447/19778

Start/stop/run cycles: 61428809/80746445/19317636
Debug(Core 0): <CompletionTag> Example #0 passed.

==========
```

6.3.4.4.4 Parallel queues – preload off – 4 scheduler threads

```
==========
This test-case uses the following configuration:
  - my_device_idx        : 0
  - my_process_idx       : 0
  - nb scheduler pdsp    : 4
  - thread #0 on pdsp    : 0
  - thread #1 on pdsp    : 1
  - thread #2 on pdsp    : 2
  - thread #3 on pdsp    : 3
  - AP private_free_queue: 1022
  - CD private_free_queue: 1023
  - hw_queue_base_idx    : 1024
  - dma_queue_base_idx   : 0
  - pool_num             : 2
Init Done

==========
data check OK!

Statistics for core 0:
Number of jobs : 137
Number of jobs per flow: 4 6 4 6 5 3 7 7 4 3 7 3 4 5 4 5 2 3 4 5 2 4 4 3 3
4 3 4 4 7 3 5
Min/mean/max processing cycles : 6161/31144/84086
Min/mean/max overhead cycles : 2715/5844/11815


Statistics for core 1:
Number of jobs : 130
Number of jobs per flow: 4 4 4 4 3 3 6 4 5 1 4 7 3 5 4 5 2 3 6 7 5 5 2 3 3
3 7 4 4 3 3 4
Min/mean/max processing cycles : 6085/33484/90643
Min/mean/max overhead cycles : 2685/6164/12383


Statistics for core 2:
Number of jobs : 128
Number of jobs per flow: 3 5 3 4 5 4 4 4 5 7 5 1 6 5 2 5 6 3 5 3 4 2 3 5 2
7 3 5 3 1 5 3
Min/mean/max processing cycles : 6145/33682/83529
Min/mean/max overhead cycles : 2677/6199/13047


Statistics for core 3:
Number of jobs : 123
Number of jobs per flow: 5 3 5 2 3 5 3 2 5 7 2 2 4 4 3 3 3 7 7 6 3 5 4 4 5
5 2 3 2 3 4 2
Min/mean/max processing cycles : 6263/34955/83293
```

```
Min/mean/max overhead cycles : 2679/6443/15181


Statistics for core 4:
Number of jobs : 127
Number of jobs per flow: 3 5 4 7 3 5 5 2 3 4 1 7 4 3 5 3 4 4 4 1 6 3 3 2 5
2 6 3 5 5 7 3
Min/mean/max processing cycles : 6313/33512/88039
Min/mean/max overhead cycles : 2681/6406/15061


Statistics for core 5:
Number of jobs : 119
Number of jobs per flow: 6 2 5 3 5 3 1 4 4 1 7 2 2 3 4 3 6 5 2 4 3 4 5 4 5
5 2 5 3 5 3 3
Min/mean/max processing cycles : 6125/35972/82947
Min/mean/max overhead cycles : 2697/6709/18799


Statistics for core 6:
Number of jobs : 137
Number of jobs per flow: 3 5 5 4 5 4 3 5 3 5 2 3 5 3 4 3 4 3 2 4 6 4 6 7 5
4 4 5 6 5 4 6
Min/mean/max processing cycles : 6163/30879/85305
Min/mean/max overhead cycles : 2687/6238/17931


Statistics for core 7:
Number of jobs : 123
Number of jobs per flow: 4 2 2 2 3 5 3 4 3 4 4 7 4 4 6 5 5 4 2 2 3 5 5 4 4
2 5 3 5 3 3 6
Min/mean/max processing cycles : 6262/34900/85341
Min/mean/max overhead cycles : 2679/6918/18907


Statistics for all cores
Total number of jobs: 1024
Total number of jobs per flow: 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
Min/mean/max processing cycles : 6085/33487/90643
Min/mean/max overhead cycles : 2677/6353/18907

Start/stop/run cycles: 61423992/80882278/19458286
Debug(Core 0): <CompletionTag> Example #0 passed.

==========
```

6.3.5 Example_0p

Example_0p is a derivative of Example_0 that implements the post-store capability of the OpenEM.

The purpose of the post-store is to offload to the infrastructure PKTDMA the transfer of local data processed by a receive function to a global memory for a post processing of these data by another entity.

The receive function allocate a local event by calling the ti_em_local() API and then call the em_send() API on this event to trigger the PKDMA transfer.

This section is to be completed.

6.3.5.1 Initialization procedure

This section is to be completed.

7 OpenEM on ARM

This section is to be completed

8 OpenEM on ARM and DSP

This section is to be completed