EVE Subsystem Starterware

User's Guide

Last updated **October 6, 2015**

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

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 relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components, which TI has specifically designated as military grade or "enhanced plastic", are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive & Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications & Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers & Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energyapps
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) starter ware implementation on the EVE Subsystem. It also provides a detailed Application Programming Interface (API) reference and information on the sample application(s) that accompanies this component.

Intended Audience

This document is intended for system engineers who want to develop vision and imaging applications using EVE subsystem.

This document assumes that you are fluent in the C language, have a good working knowledge of embedded system and basic computer architecture concepts like DMA, Cache, interrupts etc.

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction** introduces the EVE subsystem and starterware components. It also provides an overview of supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the starter ware.
- **Chapter 3 - API Reference** describes the data structures and interface functions used in the starter ware.
- **Chapter 4 - Sample Usage**, describes the sample usage of the starter ware and explains the different examples provided as part of starter ware component.
- **Chapter 5 - Frequently Asked Questions,** provides answers to few frequently asked questions related to using the starter ware.

Related Documentation From Texas Instruments

The following documents describe EVE subsystem details. To obtain a copy of any of these TI documents, visit the Texas Instruments website at www.ti.com.

- *Embedded Vector Engine (EVE) Programmer's Guide* (literature number SPRUHC1B) describes EVE architecture and all modules of EVE subsystem from programmer's view
- *EVE Subsystem Reference Guide* (literature number SPRUHF5A) describes the function description of EVE subsystem and its Register set

- *ARP32 CPU and Instruction Set* (literature number SPRUHC9) describes ARP32 architecture, Instruction Set and programming model
- Enhanced Direct Memory Access (EDMA3) Controller User's Guide (literature number SPRUEQ5) for complete details on the EDMA3
- VisionSurround28 Super/High/Mid Vision28 Super/High/Mid ADAS Applications Processor (SPRS884D)

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
ARP32	32-bit Advanced RISC Processor
DMEM	Data Memory in EVE Sub System
EVE	Embedded Vision Engine
EDMA	Enhanced Direct Memory Access
IBUF	Image Buffer in EVE Sub System
IPC	Inter-Processor Communication
MMU	Memory Management Unit
RAM	Random Access Memory
SCTM	Sub System Counter and Timer Module
SMSET	Software Message and System Event Trace
VCOP	Vision Co-Processor
WBUF	Work Buffer in EVE Sub System

Text Conventions

The following conventions are used in this document:

- Text inside back-quotes (‘’) represents pseudo-code.
- Program source code, function and macro names, parameters, and command line commands are shown in a `mono-spaced` font.

Product Support

1.1 When contacting TI for support on this eve starterware software component, quote the product name *EVE Subsystem Starter ware* and version number. The version number of the product is included in the Release Notes that accompanies this product.

Trademarks

This page is intentionally left blank

Contents

Figures

Tables

Chapter 1

Introduction

The EVE subsystem consists of a vision co-processor (VCOP), several system modules including a mailbox, a cache controller, Memory Management Unit (MMU), an EDMA controller, and an interrupt controller, plus an ARP32 scalar processor that controls all of these modules. The EVE starterware package is an OS agnostic register level API that runs primarily on the ARP32 to configure and use the features of the various EVE modules. An additional small set of APIs run on a host (DSP or ARM) for communication with the ARP32.

This chapter provides brief overview of the hardware blocks in the EVE subsystem and what the starterware offers for each block. For detailed description, please refer to EVE Programmers Guide and EVE Subsystem Reference Guide. The purpose of this User's Guide is to provide detailed information regarding the software elements and test examples provided with EVE Starterware package.

1.1. Overview of EVE Subsystem

The Embedded Vision Engine (EVE) module is a programmable imaging and vision-processing engine, intended to be used in devices that serves consumer electronics imaging and vision applications. Its programmability allows late-in-development or post-silicon processing requirements to be met, and allows third party or customers to add differentiating features in imaging and vision products. The EVE module is instantiated in Vision super/High/Mid ADAS Application Processors

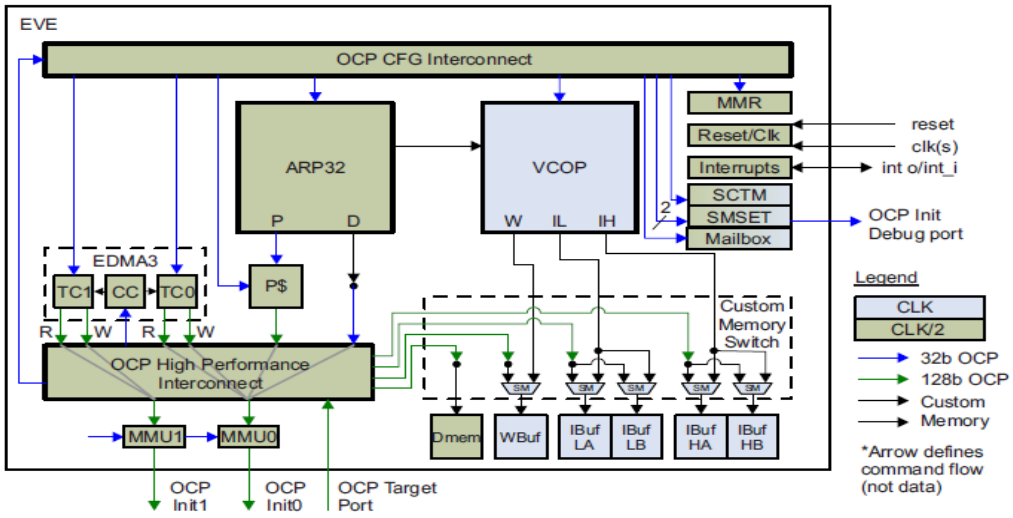

Figure 1-1. Block diagram of EVE Subsystem

As shown in Figure 1-1 there are different programmable module in EVE subsystem like EDMA, Mailbox, SCTM, MMU, Program Cache, Interrupts, ARP32 and VCOP. VCOP is main compute engine for vector processing and ARP32 is the scalar processor. Other blocks are control and data transfer modules.

1.2. Overview of EVE starterware

The EVE starterware package is an OS agnostic register level API that runs primarily on the ARP32 to configure and use the features of the various EVE control and data transfer modules. An additional small set of APIs run on a host (DSP or ARM) to configure the EVE subsystem modules. Host APIs are direct configuration of EVE subsystem modules from external processor other than the ARP32 of EVE subsystem, these are not Inter processor communication to ARP32 to configure these modules. Primary purpose of this package is to abstract low level details of EVE subsystem and share a rich example code to show case the usage of these APIs for different purpose. The starterware along with low level DMA functions, also provides high level DMA utilities to abstract most commonly required data transfer pattern in vision and imaging applications

1.3. Supported Services and Features

This user guide accompanies EVE starterware library and example source code. This version of the library has the following supported features.

- APIs to configure below modules on ARP32
  - Mailbox
  - Interrupt controller
  - MMU
  - Program Cache
  - SCTM
  - EDMA
  - EDMA Utilities
- APIs to configure below modules on DSP (C66x)
  - Mailbox
  - MMU
  - Program Cache
- Examples for below use cases
  - edma_circ1d_eve
  - edma_circ1d_rewrite_eve
  - edma_eve_double_buffering_alias_view
  - edma_eve_ROI_w_linking
  - edma_simple_eve
  - eve_bfswtch_arp32_error_intr
  - eve_bfswtch_vcop_error_intr
  - mailbox_dsp1_all_eves(vayu only)
  - mailbox_eve1_to_dsp1

- mailbox_eve1_to_dsp1_inorder_ack
- program_cache_block_inv
- program_cache_global_inv
- program_cache_software_prefetch
- ref_mmu_tlb
- ref_mmu_tlb_miss_tlw_dis
- sctm_counters_observing_8_events
- sctm_vcop_busy_time
- vcop_done_intr
- vcop_error_intr
- vcop_max_iters_intr

This page is intentionally left blank

Chapter 2

Installation Overview

This chapter provides a brief description on the system requirements and instructions for installing the starterware component. It also provides information on building and running the example application.

2.1. System Requirements

This section describes the hardware and software requirements for the normal functioning for this component.

2.1.1. Hardware

The starterware has been built and tested on the EVE subsystem based devices like TDA1MEV /Vision28 Super (Vayu). In this document vme implies TDA1MEV and Vayu implies Vision28 Super.

2.1.2. Software

Kindly refer to EVE SW getting started guide which is located at docs folder at EVE SW root directory.

2.2. Installing the Component

The starterware component is released as a windows/unix installer along with other software modules for EVE. Figure 2-1 shows the sub-directories created after installation.

> Note:
>
> The source folders under drivers are not present in case of a library based (object code) release. In this document vme implies TDA1MEV and Vayu implies Vision28 Super.

Figure 2-1. Component Directory Structure

Table 2-1 provides a description of the sub-directories created in the starterware directory.

Table 2-1. Component Directories

Sub-Directory	Description
/docs	Contains Documents related to Starterware
/drivers/devices/vayu/cred/inc	Contains CRED header files for vayu platform
/drivers/devices/vayu/cred/src	Contains CRED src files for vayu platform
/drivers/devices/vme/cred/inc	Contains CRED header files for vme platform
/drivers/devices/vme/cred/src	Contains CRED src files for vme platform
/drivers/inc	Contains internal header files needed for starterware. This folder will not be present for library based release
/drivers/src	Contains src files needed for starterware. This folder will not be present for library based release
/drivers/src/host_pc	Contains src for host/PC emulation for Starterware
/drivers/src/edma_utils	Contains src files for EDMA utility use cases
/examples/common	Contains common src and headers needed by example code
/examples/common/dsp/inc	Contains API's to configure DSP interrupts
/examples/common/dsp/src	Contains source code for DSP interrupt setupt
/examples/common/vayu	Contains boot_arp32.asm file for vayu. This is needed by all examples and contains interrupt vector mapping for ARP32
/examples/common/vme	Contains boot_arp32.asm file for vme. This is needed by all examples and contains interrupt vector mapping for ARP32
/examples	Contains examples illustrating various usage of starterware API's. More details on examples is provided in 0.
/inc/baseaddresses/vayu/eve	Contains header files which contains base addresses for eve subsystem and is sub modules for vayu platform for eve core
/inc/baseaddresses/vayu/dsp	Contains header files which contains base addresses for eve subsystem and is sub modules for vayu platform for dsp core
/inc/baseaddresses/vme/eve	Contains header files which contains base addresses for eve subsystem and is sub modules for vme platform for eve core
/inc/baseaddresses/vme/dsp	Contains header files which contains base addresses for eve subsystem and is sub modules for vme platform for dsp core
/inc	Contains API's of starterware
/inc/edma_csl	Contains header files for EDMA Chip support Library. In future, this folder is expected to be replaced with proper API's of EDMA in starterware
/inc/edma_utils	Contains header files for various EDMA utilities

Sub-Directory	Description
/libs/vayu/eve	This folder will be created once you build starterware library and contains library for eve starterware for vayu platform, which can be called from eve host.
/libs/vayu/dsp	This folder will be created once you build starterware library and contains library for eve starterware for vayu platform, which can be called from dsp host.
/libs/vme/eve	This folder will be created once you build starterware library and contains library for eve starterware for vme platform, which can be called from eve host.
/libs/vme/dsp	This folder will be created once you build starterware library and contains library for eve starterware for vme platform, which can be called from dsp host.
/STMlib	This folder contains source and API's which can be used by for using SMSET functionality. Currently this is not supported by Code Composer Studio on vayu platform. For details on its usage please refer SMSET chapter of EVE's Programmer guide .

2.3. Building Starterware Libraries

Starterware Libraries can be found at libs folder present in top most folder in starterware directory. This libs folder contains four libraries of starterware depending on platform and host.

Platform : vayu Host: eve

Platform : vayu Host: dsp

Platform : vme Host: eve

Platform : vme Host: dsp

Starterware build uses GNU make system for building. For Building Starterware first you will need to set following four environment variable in your system:

ARP32_TOOLS : Directory pointing to ARP32 compiler (can be found inside CCS installation at following location <CCS_INSTALLATION_DIR>/ccsv5/tools/compiler/)

DSP_TOOLS : Directory pointing to DSP compiler (can be found inside CCS installation at following location

<CCS_INSTALLATION_DIR >/ccsv5/tools/compiler/)

EVE_SW_ROOT : Directory Pointing to EVE softwares root directory

UTILS_PATH : Directory pointing to utils command like mkdir, rm (can be found inside CCS installation at following location on windows :

<CCS_INSTALLATION_DIR>/ccsv5/utils/cygwin

On linux this directory is same as /bin

Once you set the above-mentioned four variables, you are all set to build starterware libraries. For building we use gmake command provided by GNU make.

Gmake command can take three inputs for building different libraries for different core and different platform. Following are the three variables:

TARGET_SOC: this is to specify the platform for which library is required to be build. Currently it can take two values: vayu and vme, for building on vayu platform or on vme platform respectively.

CORE: this is to specify the host for which library is required to be build. Currently it can take two values: eve and dsp, for building for eve host or dsp host respectively.

TARGET_BUILD: this is to specify whether to build library in release mode or debug mode. It can take two values: release and debug.

Rules.make file will configure various variables needed for build based on these three inputs. If you do not give any inputs by default, these variables will take following values: TARGET_SOC = vayu, CORE=eve and TARGET_BUILD=release.

For building EVE starterware library for vayu platform and eve as host:

gmake TARGET _SOC=vayu CORE=eve

Alternatively, you can use just gmake for this particular case as this case uses default.

Library will be created and placed at:

<STARTERWARE_DIR>/libs/vayu/eve/libevestarterware_eve.lib

Building EVE starterware library for vayu platform and dsp as host :

gmake TARGET_SOC=vayu CORE=dsp

Alternatively, you can use just gmake CORE=dsp for this particular case as this case TARGET_SOC is same as default.

Library will be created and placed at:

<STARTERWARE_DIR>/libs/vayu/dsp/libevestarterware_dsp.lib

Similarly, you can build library for vme platform also, final libraries will be located at following places:

<STARTERWARE_DIR>/libs/vme/eve/libevestarterware_eve.lib

<STARTERWARE_DIR>/libs/vme/dsp/libevestarterware_dsp.lib

2.4. Building Starterware Examples

Starterware examples also use similar procedure as described above. Each example contains its own Makefile present in src folder of respective examples directory. Some examples needs hosts for configuration, those example contains folder for host (dsp) and eve. By default each example is built in debug mode only. If you want to build it in release mode change it in make file.

This page is intentionally left blank

Chapter 3

API Reference

This chapter provides a detailed description of various API provided for sub module like Cache, EDMA, mailbox, mmu, interrupts etc by eve starterware

1.1. Mailbox

EVE has an internal mailbox to support synchronization and message passing between ARP32 and system level hosts. The mailbox function supports 2-way communication between maximum 4 users. This function relies on internal sub-modules, each supporting 1-way communication between one user referred to as the sender, and another user referred to as the receiver. The allocation of the mailbox sub-module to the communication between two users is done by software. Whenever a message is written in the appropriate sub-module, the associated interrupt is sent to the appropriate receiver, provided the interrupt is enabled. Each interrupt has its own interrupt status register and interrupt enable register. The line interrupt signal indicates when one or more events are detected by the hardware, for the corresponding user. Each event is independently maskable. Since interrupt status register are independent from one user to another, the associated interrupt line behavior are independent from each other as far as user doesn't access not owned mailbox. Please refer to EVE programmer's guide for more details

API NAME	**Description**
EVE_MBOX_Read	Read a value from the specified mailbox
EVE_MBOX_Write	Write value to specified mailbox
EVE_MBOX_Reset	Soft reset of mailbox
EVE_MBOX_IsFull	Check if mailbox is full
EVE_MBOX_GetNumMsg	Get number of messages in mailbox
EVE_MBOX_IrqEnable	Enable interrupt for user/mailbox
EVE_MBOX_IrqDisable	Disable interrupt for user/mailbox
EVE_MBOX_IrqGetStatusAll	Get interrupt status for all users of a mailbox
EVE_MBOX_IrqGetStatus	Get interrupt status for a specific user
EVE_MBOX_IrqClearStatus	Clears interrupt status for user/mailbox

Paths:

Sources- starterware/drivers/src/mbox.c

Prototypes - starterware/inc/mbox.h

Library- starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_eve.lib

starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_dsp.lib

1.2. Interrupt Controller

The interrupt controller handles incoming interrupts, merging them with internal interrupt sources to drive ARP32's interrupt inputs. The interrupt controller also allows ARP32 to generate outgoing interrupts or events to synchronize with system ARM, DSP, and EDMA. It supports up to 32 active-high level interrupt inputs, and outputs 5 active high level interrupt outputs. Its architecture allows both hardware and software prioritization. Please refer to EVE programmer's guide for more details.

API NAME	**Description**
EVE_INTCTL_LevelInit	Configure the priority and interrupt kind
EVE_INTCTL_OneITEnable	Enable the specified interrupt
EVE_INTCTL_AllITEnable	Enable all the interrupts
EVE_INTCTL_Ack	Read the interrupt status register
EVE_INTH_InterruptEnable	Enable one interrupt
EVE_INTH_InterruptDisable	Disable one interrupt

Paths:

Sources – starterware/drivers/src/INTCTL.c

Starterware/drivers/src/INTH.c

Prototypes – starterware/inc/INTH.h

Library- starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_eve.lib

starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_dsp.lib

1.3. Cache Controller

The Program Cache in EVE is always enabled, and all ARP32 fetch accesses are cacheable. For cache hits, the interface between ARP32 and Program Cache handles back to back requests with 0 cycle latency in order to provide full throughput/program execution for the ARP32. For Cache Misses, the Program Cache controller will stall the ARP32 until the return of the cache line. The Program Cache also contains a software and performance transparent 256-b line buffer to minimize power consumption by minimizing accesses to the underlying Cache/SRAM in the case of back to back hits to the same line. Please refer to EVE programmer's guide for more details.

API NAME	**Description**

EVE_PROGCACHE_BlockInvalidate	Program cache block invalidate
EVE_PROGCACHE_GlobalInvalidate	Program cache global invalidate
EVE_PROGCACHE_Prefetch	Program cache pre-fetch

Paths:

Sources – starterware/drivers/src/pcache.c

Prototypes – starterware/inc/pcache.h

Library- starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_eve.lib

starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_dsp.lib

1.4. MMU

There are two MMUs that are accessible via the interconnect loopback path. Each of the two TCs is mapped to one of the two MMUs. One of the MMUs is also shared with ARP32 program and data accesses. The ARP32 Program Cache and Data accesses share an MMU in order to provide a convenient mechanism whereby ARP32 debug accesses have a single/consistent view of the system that is equivalent to ARP32 software's view. For the EDMA paths, the two MMUs are required to provide maximum concurrency for each TC and its respective accesses to system memory.

API NAME	**Description**
EVE_MMU_CurrentVictimSet	Set current victim
EVE_MMU_CurrentVictimGet	Get current victim
EVE_MMU_TlbLockSet	Set base lock value
EVE_MMU_TlbLockGet	Get base lock value
EVE_MMU_GlobalFlush	MMU global flush
EVE_MMU_TlbEntryFlush	MMU TLB entry flush
EVE_MMU_SoftReset	Reset MMU
EVE_MMU_IrqGetStatus	Get MMU interrupt status
EVE_MMU_IrqClearStatus	Clear MMU interrupt status
EVE_MMU_IrqEnable	Enable MMU interrupt
EVE_MMU_IrqDisable	Disable MMU interrupt
EVE_MMU_WtlEnable	Enable walking table logic
EVE_MMU_WtlDisable	Disable walking table logic
EVE_MMU_Enable	Enable MMU
EVE_MMU_Disable	Disable MMU
EVE_MMU_TlbEntrySet	Configure selected TLB entry

Paths:

Sources – starterware/drivers/src/mmu.c

Prototypes – starterware/inc/mmu.h

Library- starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_eve.lib

starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_dsp.lib

1.5. Memory Switch and Mapping

The custom memory switch provides a statically multiplexed low latency/high bandwidth switch between a master (VCOP/System/EDMA/ARP32) and internal EVE memories. Based on the value in the MMR register buffer ownership register programmed by the RISC core, either the VCOP or the System is designated as the master of the memory. The ARP32 read/write accesses and EDMA accesses are multiplexed and designated as the System master. Buffer switch MMRs are configured by ARP32 during run-time. Memory block structure is chosen in hardware to facilitate concurrent ping/pong DMA vs. processing, where the EDMA/ARP32 (System) may own one block of memory and VCOP may own another block of memory.

These are simple operations for which no APIs are provided. The applets 2.1.4 and 2.1.5 in 'examples' folder illustrates the use of buffer switching.

1.6. SCTM (Sub System Counter and Timer Module)

The SCTM module provides a specification for a generic counter timer module that can be instantiated within the processor subsystem and functions as a centralized profiling module for the entire subsystem. This module will map a large number of system /subsystem event signals to a smaller number of counter resources, controlled by user software. Some of the counter resources in the module can be configured for timer functionality where system and/or debug events can be generated when designated intervals are matched. It provides a unified programmers view of all profiling resources within the subsystem, which is accessible by either debug tools and/or the application. In addition to profiling functions, resources in this module address the need for application counter/timer functions such as OS timers to generate periodic interrupts or schedule periodic tasks.

API NAME	**Description**
EVE_SCTM_Enable	Enable SCTM hardware
EVE_SCTM_CounterReset	Reset SCTM counter
EVE_SCTM_ChainModeEnable	Enable 2 counters in chained 64-bit mode
EVE_SCTM_CounterTimerEnablev	Enable counter working as a timer
EVE_SCTM_CounterTimerDisable	Disable counter working as a timer
EVE_SCTM_MultipleCountersEnable	Enable multiple counters at once in sync
EVE_SCTM_MultipleCountersDisable	Disable multiple counters at once in sync
EVE_SCTM_TimerInterruptEnable	Enable timer to generate interrupt
EVE_SCTM_CounterRead	Read counter value
EVE_SCTM_OverflowCheck	Check for overflow in counter
EVE_SCTM_CounterConfig	Configure counter

EVE_SCTM_CounterChainModeConfig	Configure counter in chain mode
EVE_SCTM_TimerConfig	Configure timer

Paths:

Sources – starterware/drivers*/src/eve1/sctm.c*

Prototypes – starterware/inc /sctm.h

Library- starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_eve.lib

starterware/libs/$(PLATFORM)/$(CORE)/ libevestarterware_dsp.lib

1.7. SMSET (Software Message and System Event Trace)

The SMSET is a trace module used for monitoring key system events, and transferring software messages. SMSET can be used by programmers to understand the performance of EVE at a macro level. While SCTM is used to drill down on performance of specific loops, SMSET can be used to monitor performance of complete tasks, and perform in system monitoring, that can then be viewed as a performance log. Please refer to EVE programmer's guide for more details.

API NAME	**Description**
STMXport_open	Opens a physical STM channel
STMXport_printf	Prints a statically formatted string
STMXport_logMsg	Prints 32-bit integers, using a formatted string
STMXport_putWord	Transports a single 32-bit value
STMXport_putShort	Transports a single 16-bit value
STMXport_putByte	Transports a single 8-bit value
STMXport_getBufInfo	Check Buffer status for number of messages queued
STMXport_flush	Flush all buffered or pending data
STMXport_getVersion	Get revision number of library
STMXport_close	Closes the STM channel

Paths:

As STM HW block is present in several processors, it is maintained as a separate library.

Sources – starterware/drivers/src/StmLibrary.c

Prototypes – starterware/STMLib/include/StmLibrary.h

Library – starterware/STMLib/lib/stm_eve_elf.lib

1.8. EDMA

The EDMA3 is the primary DMA engine for transfers between system memory (DDR and/or L3 SRAM) and EVE internal memories. Channel controller (CC) is the front end programmer interface,

to the DMA engine. It is a small processor that accepts a program from a memory region called PaRAM (parameter memory) to describe the attributes of the data transfer and efficiently calculates new addresses for regular data transfers. Transfer controller (TC) is an actual physical channel that services the requests programmed on the Channel controller. The transfer controller aspect is transparent to the user/programmer, hence user need not worry about it apart from knowing that EVE subsystem EDMA has two TC channels (in TDA2x device) to ensure maximum concurrency for data transfers along with computation. Mapping of a transfer to particular TC is programmable.

EDMA API's are divided into two parts: One for allocating the resources (present in dma_resource_allocator.h) and other for configure EDMA registers (present in dma_funcs.h). Apart from these basic API's, starterware also provide some special use-cases implemented using EDMA as a part of utility offering of Starterware.

dma_resource_allocator.h

API NAME	**Description**
DMA_resourceAllocator_initResources	This functions initializes resets all the allocated resources of EDMA. After this call all the EDMA resources are freshly available
DMA_resourceAllocator_allocateResources	This functions allocates EDMA resources for given number of channels. It also needs to know the EDMA attribute (EDMA/QDMA channel). How many param sets needed for each channel and which hardware que each channel should go to.
DMA_resourceAllocator_deallocateResources	This functions deallocates EDMA resources which are allocated using DMA_resourceAllocator_allocateResources API

dma_functions.h

API NAME	**Description**
DMA_funcs_hardwareRegSetup	This function updates the actual hardware register based on DMA_resourceStruct. For QDMA we will be setting up QCHMAP, QDMAQNUM, QEESR. Similarly for EDMA we will be setting up, DCHMAP, DMAQNUM. Independent of QDMA and EDMA we configure QUEWTHRA, QUEPRI and QUETCMAP registers. This implementation assumes one to one mapping between transfer controllers and hardware queues. TC0 is mapped to Q0 and TC1 is mapped to Q1, this means Q0 submits on TC0 and Q1 submits on TC1. More over Q0 is given a higher priority compare to Q1.
DMA_funcs_writeTransferParams	Updates paramset entry based on user input
QDMA_SUBMIT	This function is used to trigger QDMA transfer. It's user of this functions responsibility to provide the address of the trigger word writing at which will trigger the QDMA transfers. This function is only aplicable for QDMA channels and should not be used for EDMA channels
QDMA_WAIT	This function waits for the completion of QEDMA transfers indicated by bit position in wait word. It also clears the IPR register by writing to 1 to the bits position indicated by wait word

DMA_SUBMIT	Submit the DMA on this virtual handle
DMA_WAIT	Wait for the DMA to complete, by polling
pack2	Helper function to pack two 16-bit values into a single 32 bit entry

All EDMA example uses edma_csl API's which are present in edma_csl folder in starterware inc directory.

Paths:

Sources – dma_funcs.c in starterware/drivers/src

dma_resource_allocator.c in Starterware/drivers/src/

Prototypes – dma_funcs.h in starterware/inc

dma_resource_allocator.h in staterware/inc

dma_resource.h in Starterware/inc

1.9. EDMA Utility

EDMA utility is Starterware offering to ease the EDMA programming from user . Idea behind EDMA utility is to provide utility functions for the most common EDMA data flow usecases in order to hide EDMA programming from the user.

Most of the vision and imaging algorithms are compute intensive and hence it is always desirable to overlap compute intensive part with data transfer so as to fully utilize both the hardware resources in parallel. It has been observed that most of the vision algorithms are implemented with some common set of data transfer's . EDMA utilities provide utility functions for such common EDMA data flow's which are generally used in many vision algorithms.

Following are the most common EDMA data flow usecases used in vision algorithms:

- Auto-Increment : Horizontal
- Auto-Increment : Vertical
- Auto-Increment : 1D (with circular buffering)
- Scatter Gather
- Blocking/Non-Blocking memcpy

Following are the set of API's which are expected to be provided for each utility:

- EDMA_UTILS_setEdma3RmHandle : This API should be called first for a set of Utility functions used inside an applet. Users can provide handle as NULL if they don't want to use EDMA3 LLD
- EDMA_UTILS_globalReset : This API should be called once per frame. This will reset all the utilities state. It is important that this should only be called once for IN or OUT transfer.
- EDMA_UTILITY_<usecase>_getContextSize: To request the utility to give the size of internal context used by it. It is always advisable to allocate the utility context in internal memory for better performance

- EDMA_UTILITY_<usecase>_init : To initialize the usecase internal context with user provided transfer property for a specific usecase
- EDMA_UTILITY_<usecase>_update: Update transfer property of usecase
- EDMA_UTILITY_<usecase>_configure : Actually configure EDMA hardware based on usecase requirement
- EDMA_UTILITY_<usecase>_deconfigure : Release the EDMA resources allocated during configure call of a usecase
- EDMA_UTILITY__<usecase>_trigger : Trigger the DMA transfer for particular usecase
- EDMA_UTILITY_<usecase>_wait : Wait for transfer to complete

There can be some additional API's for separating the INPUT and OUTPUT transfers for trigger and wait. The reason for keeping them separate is to remove overheads of checking whether its INPUT transfer or OUTPUT transfer

Sources – EDMA_UTILS_<utilityname>.c in starterware/drivers/src/edma_utils

Prototypes – EDMA_UTILS_<utilityname>.h in starterware/inc/edma_utils

1.9.1. EDMA Utility Autoincrement

Auto-increment Data flow means that every time user triggers a DMA transfer it should fetch a block without user/CPU intervention. All the addresses updating is hidden from the user and is either done by EDMA hardware itself or done by utility for the cases where it cannot be done by hardware. EDMA autoincrement utility provides init/configure function which will configure hardware once and later user is expected to only call EDMA trigger and EDMA wait. **Figure 3-1**displays basic horizontal auto-increment. Note that blocks need not be overlapping as it is shown in the figure and also the internal memory organization is not necessary to be same.

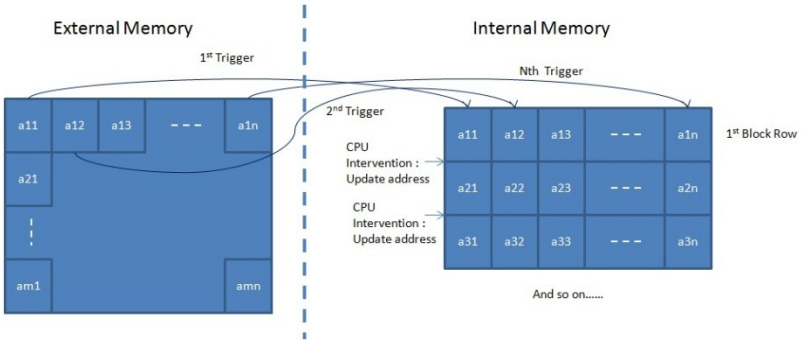

Figure 3-1. Autoincrement Usecase

Auto-increment can also be of two types:

- Horizontal auto-increment: In this usecase, blocks are incremented in horizontal direction first followed by vertical direction.
- Vertical auto-increment: In this usecase, blocks are first incremented in vertical direction then in horizontal direction.

This usecase is most common data flow, which is needed by vision algorithms. For example any filtering operation generally works on blocks which are incremented either in horizontal or vertical direction. In the subsequent section, we will be describing various autoincrement utilities that are offered by EVE Starterware.

1.9.1.1. EDMA_UTILITY_AUTOINCREMENT

This is the default autoincrement usecase and it provides EDMA utility implementation for 2D horizontal auto-increment usecase. This utility handles both overlapping and non-overlapping increments. Lets look at the interface for this utility.

Following is the data type for configuring EDMA_AUTO_INCREMENT utility

```
typedef struct
{
    uint16_t    roiWidth;
    uint16_t    roiHeight;
    uint16_t    blkWidth;
    uint16_t    blkHeight;
    uint16_t    extBlkIncrementX;
    uint16_t    extBlkIncrementY;
    uint16_t    intBlkIncrementX;
    uint16_t    intBlkIncrementY;
    uint32_t    roiOffset;
    uint8_t     *extMemPtr;
    uint8_t     *interMemPtr;
    uint16_t    extMemPtrStride;
    uint16_t    interMemPtrStride;
    uint8_t     dmaQueNo;
}EDMA_UTILS_autoIncrement_transferProperties;
```

This structure specifies the properties of the transfer for auto increment usecase. It is important to note the region of interest should only include the valid transfer region. This utility assumes that following equation is satisfied :

(roiWidth - blkWidth) % extBlkIncrementX = 0

(roiHeight - blkHeight) % extBlkIncrementY = 0

roiWidth : Width of the region of interest in an image frame. User should be careful that for non-overlapping increment usecase roiWidth should not include the last increment in X direction in order to satisfy the above stated equation.

roiHeight : Height of the region of interest in an image frame. User should be careful that for non-overlapping increment usecase roiHeight should not include the last increment in Y direction.

blkWidth : Block Width

blkHeight : Block Height

extBlkIncrementX : Block Increment in X direction for external memory. If extBlkIncrementX = 0 then number of blocks for which autoincrement with run in horizontal directions is 1.

extBlkIncrementY: Block Increment in Y direction for external memory.If extBlkIncrementY = 0 then number of blocks for which autoincrement with run in vertical directions is 1.

intBlkIncrementX : Block Increment in X direction for internal memory

intBlkIncrementY : Block Increment in Y direction for internal memory

roiOffset : Offset from the base pointer of the image to the point from where ROI starts

extMemPtr : Pointer to external memory buffer

interMemPtr :Pointer to internal memory buffer. When using autoincrement utility from BAM user need not provide this pointer as it is internally allocated by BAM and automatically gets set during setMemRec function

extMemPtrStride : Stride/Pitch for external memory buffers

interMemPtrStride: Stride/Pitch for internal memory buffers

dmaQueNo: Queue/TC number to be used (0 or 1)

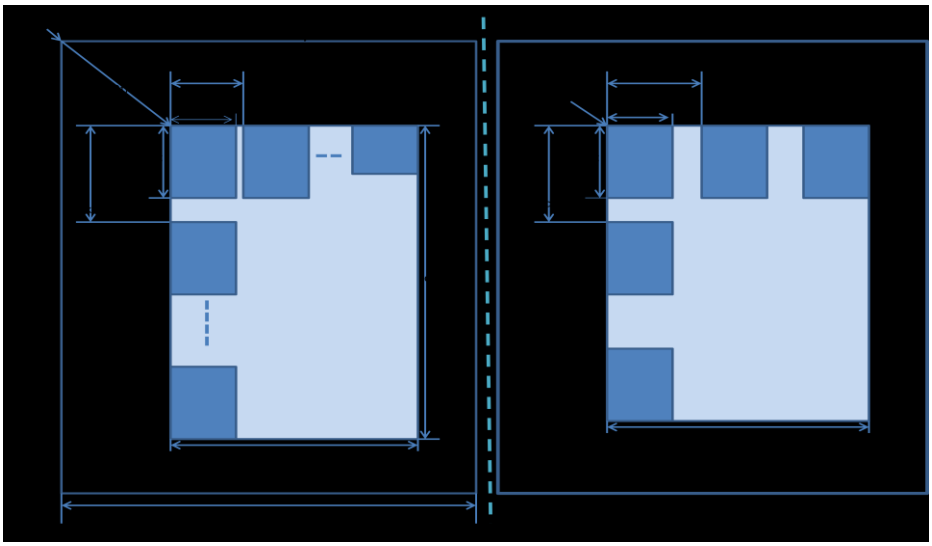

Figure 3-1. Autoincrement Utility Init parameters

API NAME	**Description**
EDMA_UTILS_autoIncrement_getContextSize	Returns the size needed by the edma utils autoincrement internal context structure. If you want to statically allocate the memory for this context you can also get the size of context from the edma_utils_context_size.h header file located at inc/edma_utils folder.
EDMA_UTILS_autoIncrement_init	Initializes the EDMA autoincrement context based on the user provided initialization parameters
EDMA_UTILS_autoIncrement_configure	This API actually configures EDMA with autoincrement init params provided during init time. This API can be called separately for IN and OUT transfers or for both INOUT transfer. It is expected that you configure IN transfers followed by OUT transfers.
EDMA_UTILS_autoIncrement_deconfigure	This API release all the EDMA resources which were requested duing EDMA_UTILS_autoIncrement_configure call. This API can be called separately for IN and OUT transfers or for both INOUT transfer. It is expected that you deconfigure IN transfers followed by OUT transfers.

EDMA_UTILS_autoIncrement_update	This API is an optional API which can be called to update the external memory address.
EDMA_UTILS_autoIncrement_triggerInChannel	This API triggers all the input transfers
EDMA_UTILS_autoIncrement_triggerOutChannel	This API triggers all the output transfers. This API also returns the status of 1 when the second last block is reached in autoincrement. This condition should be used to come out of while loop. First time call to this function will not result any dma transfer and is more like a dummy call.
EDMA_UTILS_autoIncrement_waitInChannel	This API waits for all the input transfers to complete. This is a blocking call
EDMA_UTILS_autoIncrement_waitOutChannel	This API waits for all the output transfers to complete. This is a blocking call

Following is the expected order in which the following API's are expected :

- First call either call EDMA_UTILS_autoIncrement_getContextSize to get the context size or read it from *edma_utils_context_size.h* (incase user wants to statically allocate this memory) . User is then expected to use this to allocate memory for autoincrement context. For all further API's this context should be given as an input argument

Note: It is advisable to allocate the auto increment context size in EVE internal memory(DMEM) for performance reasons. This is because this context is read multiple types for every block in the autoincrement loop and hence keeping it in internal memory will have better performance.

- Once context is allocated you are ready to initialize auto increment utility with the transfers you want to configure. This can be done using EDMA_UTILS_autoIncrement_init API. Note that this API can be called separately for IN and OUT transfers or together for INOUT transfer. It is expected that IN transfers are configured first followed by OUT transfers when we are initializing IN and OUT channels separately

- There is an optional API provided to update DDR/external memeory pointer for the cases when you don't have these addresses available during init time. The API to update external memory pointers is EDMA_UTILS_autoIncrement_update

- Once you have initialized the autoincrement context with the transfers you need then you can use this API to configure autoincrement which internally programs EDMA registers with the user provided configuration

- After the above steps we are ready to start autoincrement. Use trigger and wait API's to trigger and wait for all transfers.

- EDMA autoincrement uses EDMA_UTILS_autoIncrement_triggerOutChannel to communicate the arrival of second last block. It returns 1 when it reaches the second last block. This condition should be used as an exit condition to come out of the autoincrement loop.

1.9.1.2. EDMA_UTILITY_AUTOINCREMENT_1D

This EDMA utility specifically handles 1D auto-increment usecase. Another feature in which this utility is different from 2D auto-increment is the fact that it supports circular buffering. Most of the interface and usage for this utility is same as EDMA_UTILITY_AUTOINCREMENT. Kindly refer to that for its usage.

Following is the data type for configuring EDMA_AUTO_INCREMENT utility

```
/** ============================================================
 *  @name   EDMA_UTILS_autoIncrement1D_transferProperties
 *
 *  @desc   This structure specifies the properties of the transfer for
 *          auto increment usecase.
 *
 *  @field totalLength
 *             Total Length of the 1D data in which auto increment is expected
 *             to rum
 *
 *  @field t numBytes
 *             Number of bytes of 1D data to be transferred per DMA trigger
 *
 *  @field extMemIncrement
 *              Increment in external memory. If extMemIncrement = 0
 *              then number of segments to be transfered is 1
 *
 *
 *  @field intMemIncrement
 *              Increment in internal memory.
 *
 *  @field extMemPtr
 *              Pointer to external memory buffer
 *
 *  @field interMemPtr
 *              Pointer to internal memory buffer.
 *              When using this autoincrement utility from BAM user need not provide
 *              this pointer as it is internally allocated by BAM and automatically
 *              gets set during setMemRec function
 *
 *  @field numCircBuf
 *              Number of buffers for circular buffering
 *
 *  @field dmaQueNo
 *              DMA Que number to which particular transfer is expected to go
 *
 *  ============================================================
 */
typedef struct
{
    uint32_t    totalLength;
    uint16_t    numBytes;
    uint16_t    extMemIncrement;
    uint16_t    intMemIncrement;
    uint8_t     *extMemPtr;
    uint8_t     *interMemPtr;
    uint8_t     numCircBuf;
    uint8_t     dmaQueNo;
}EDMA_UTILS_autoIncrement1D_transferProperties;
```

This utility also expects the following equation to hold true :

(totalLength – numBytes) % extMemIncrement = 0

API NAME	Description
EDMA_UTILS_autoIncrement1D_getContextSize	Returns the size needed by the edma utils autoIncrement1D internal context structure. If you want to statically allocate the memory for this context you can also get the size of context from the edma_utils_context_size.h header file located at inc/edma_utils folder.
EDMA_UTILS_ autoIncrement1D _init	Initializes the EDMA autoIncrement1D context based on the user provided initialization parameters
EDMA_UTILS_ autoIncrement1D _configure	This API actually configures EDMA with autoIncrement1D init params provided during init time. This API can be called separately for for IN and OUT transfers or for both INOUT transfer. It is expected that you configure IN transfers followed by OUT transfers.
EDMA_UTILS_ autoIncrement1D _deconfigure	This API release all the EDMA resources which were requested duing EDMA_UTILS_autoIncrement1D_configure call.. This API can be called separately for for IN and OUT transfers or for both INOUT transfer. It is expected that you deconfigure IN transfers followed by OUT transfers.

EDMA_UTILS_ autoIncrement1D _update	This API is an optional API which can be called to update the external memory address.
EDMA_UTILS_ autoIncrement1D _triggerInChannel	This API triggers all the input transfers
EDMA_UTILS_ autoIncrement1D _triggerOutChannel	This API triggers all the output transfers. This API also returns the status of 1 when the second last block is reached in autoincrement. This condition should be used to come out of while loop. First time call to this function will not result any dma transfer and is more like a dummy call.
EDMA_UTILS_ autoIncrement1D _waitInChannel	This API waits for all the input transfers to complete. This is a blocking call
EDMA_UTILS_ autoIncrement1D _waitOutChannel	This API waits for all the output transfers to complete. This is a blocking call

This utility can be used to do ping/pong or circular buffering even with out using ALIAS view of EVE memory.

1.9.2.EDMA_UTILS_memcpy2D

This is another EDMA utility provided by EVE Starterware. This is actually generic 2D memcpy. This utility is different from all other utility in a sense that which is a stand alone utility and it doesn't need any initialization and can be called from anywhere. This utility internally uses dedicated EDMA resources. It is to be noted that this utility is a blocking call. Following is the description of various parameters which user needs to be provide for this utility.

```
/* ================================================
 *  @func     EDMA_UTILS_memcpy2D
 *
 *  @desc     This function used EDMA module of eve subsytem to do a 2D memcpy
 *
 *  @modif
 *
 *  @inputs   This function takes following Inputs
 *                  dstPtr :
 *                          Pointer to destination
 *                  srcPtr :
 *                          Pointer to source
 *                  width :
 *                          width of 2D block to be transfered
 *                  height :
 *                          height of 2D block to be transfered
 *                  dstStride :
 *                          Stride/Pitch for dstPtr
 *                  srcStride :
 *                          Stride/Pitch for srcPtr
 *
 *  @outputs NONE
 *
 *  @return    0 : Success
 *                  -1 : Failure
 *
 *  ================================================
 */

int32_t EDMA_UTILS_memcpy2D(void * dstPtr, void * srcPtr , uint32_t width, uint32_t height,
                            int32_t dstStride, int32_t srcStride);
```

1.9.3.EDMA_UTILITY_SCATTERGATHER

Another popular usecase which is commonly used in vision algorithms is scatter gather usecase. In Scatter gather data flow, input blocks for processing are scattered across the whole image without any periodicity. It is expected that every trigger will fetch these scattered block to a particular memory. It is to be noted that this utility need not expect that all blocks are gathered at one location only. Instead it is flexible in a sense that even in destination pointers for each block could be scattered across the image.

Following is the data type for describing a single transfer for configuring EDMA_SCATTERGATHER utility. Each field in this structure is a list of entries describing transfer property for the complete list of transfer that needs to be done in one trigger

```
/** ============================================================
 *  @name   EDMA_UTILS_scattherGather_transferProperties
 *
 *  @desc   This structure specifies the properties of the transfer for
 *          scatter gather usecase.
 *
 *  @field updateMask
 *              Mask of fields telling which fields needs to be updated.
 *              Refer to EDMA_UTILS_SCATTERGATHER_UPDATE_TYPE for valid
 *              values. User can provided more than one field to be updated
 *              by ORing the above enum. This field is a don't care(ignored) during
 *              EDMA_UTILS_scattherGather_init
 *
 *  @field dmaQueNo
 *              DMA Que number to which particular transfer is expected to go.
 *              This is dont care (ignored) during EDMA_UTILS_scattherGather_updateNtrigger
 *
 *  @field srcPtr
 *             Pointer to the list of source pointer
 *
 *  @field t dstPtr
 *            Pointer to the list of destination pointer
 *
 *  @field srcPtrStride
 *              Pointer to the list of stride for the source pointer
 *
 *  @field dstPtrStride
 *              Pointer to the list of stride for the destination pointer
 *
 *  @field blkWidth
 *              Pointer to the list of Block width for the transfer
 *
 *  @field blkHeight
 *              Pointer to the list of  Block Height for the transrer
 *
 *  ============================================================
 */
typedef struct
{
    uint8_t     updateMask;
    uint8_t     dmaQueNo;
    uint8_t     **srcPtr;
    uint8_t     **dstPtr;
    uint16_t    *srcPtrStride;
    uint16_t    *dstPtrStride;
    uint16_t    *blkWidth;
    uint16_t    *blkHeight;
}EDMA_UTILS_scattherGather_transferProperties;
```

API NAME	Description
EDMA_UTILS_scattherGather_getContextSize	Returns the size needed by the edma utils scattergather internal context structure. If you want to statically allocate the memory for this context you can also get the size of context from the edma_utils_context_size.h header file located at inc/edma_utils folder.
EDMA_UTILS_scattherGather_init	This function configures EDMA hardware based initParams. It is important to note that this particular utility configures the hardware during this call instead of configure call
EDMA_UTILS_scattherGather_deinit	This function releases all the allocated EDMA resources for this particular usecase.
EDMA_UTILS_scattherGather_configure	Dummy function
EDMA_UTILS_scattherGather_updateNtrigger	This function can selectively update four properties of the transfers and actually trigger the transfer. These prioperties are are sourceAddress, Destination Address, block widht and block Height. Which property needs to be updated can be given in EDMA_UTILS_scattherGather_updateParams. Refer this structure for futher details on each

	individual fields. This function can only be called after EDMA_UTILS_scattherGather_init has been called. Size of array update param should be same as what has already been initialized in EDMA_UTILS_scattherGather_init.
EDMA_UTILS_scattherGather_updateSrcNtrigger	This function updates the source pointer for all the transfers and trigger the transfer. This function should be used when only source pointer is getting updated after initializing other parameters. This function can only be called after EDMA_UTILS_scattherGather_init has been called. Size of array update param should be same as what has already been initialized in EDMA_UTILS_scattherGather_init
EDMA_UTILS_scattherGather_updateDstNtrigger	This function updates the destination pointer for all the transfers and trigger the transfer. This function should be used when only destination pointer is getting updated after initializing other parameters. This function can only be called after EDMA_UTILS_scattherGather_init has been called. Size of array update param should be same as what has already been initialized in EDMA_UTILS_scattherGather_init
EDMA_UTILS_scattherGather_wait	This function waits for the DMA transfer to be completed and can be used for any of the above trigger.

Following is the expected order in which the following API's are expected :

- First call either call EDMA_UTILS_scatterGather_getContextSize to get the context size or read it from *edma_utils_context_size.h* (incase user wants to statically allocate this memory) . User is then expected to use this to allocate memory for this utility. For all further API's this context should be given as an input argument

Note: It is advisable to allocate the scatterGather context size in EVE internal memory(DMEM) for performance reasons. This is because this context is read multiple types for every block in the autoincrement loop and hence keeping it in internal memory will have better performance.

- Once context is allocated you are ready to initialize scatter gather utility with the transfers you want to configure. This can be done using EDMA_UTILS_scatterGather_init API.This API actually configures EDMA hardware based on init parameters

- After initialization it will depend on the usecase to decide on which trigger API to use. Any of the following trigger API can be used :

EDMA_UTILS_scattherGather_updateNtrigger

EDMA_UTILS_scattherGather_updateSrcNtrigger

EDMA_UTILS_scattherGather_updateDstNtrigger

- To wait for the completion of transfer use EDMA_UTILS_scattherGather_wait API.

This page is intentionally left blank

Chapter 4

Sample Usage

This chapter provides a detailed description of the sample test examples that accompanies this eve starterware component

2.1. Overview of the Example Application

The example applications are present in examples folder in starterware repository. This folder contains examples describing usage of various eve starterware sub modules like mailbox, interrupt controller, MMU, EDMA, SCTM etc. Build procedure for building examples is same as the one described in section 2.4. Each example contains its own Makefile. These make file in turn are dependent on Rules.make present at the starterware directory inside EVE software component and common.mk (present in examples/common/common.mk). These make files populate variables needed to build example codes. Example code directory structure is as shown in the Figure 2-1.

Note: All mailbox examples expects that interrupts from EVE sub systems are connect to the DSP interrupts line. For vayu platform this can be done by configuring the crossbar properly. Cross bar can be configured in multiple ways. For these examples we are using gel files to configure crossbar. Gel files for configuring crossbars is located at modules/gels/<PLATFORM> folder. For vayu platform you should use xbar_config.gel to configure cross bar. This gel file should be run from A15 before connecting to EVE. Kindly refer to Starterware/examples/common/examples_platform.h file to see figure out the interrupt number used for mail boxes.

For VME there are two gel files located in modules/gels/vme. DM814x_EVM_PG2_1_HDMI.gel should be run from A8 and eden_arp32.gel should be run from ARP32.

2.1.1.Mailbox_eve1_to_dsp1

The mailbox hardware can be used to sustain unidirectional communication. In order to do this, we need to write code on both the DSP and the EVE. In this example, we model EVE as the sender of messages, and DSP as the receiver of the messages. Further, we want to model, asynchronous nature of messages, where the sender can send messages without the receiver, immediately reading those messages, as it may be busy with actual processing.

This example consists of two programs, one running on EVE and other running on DSP. In EVE side, we first configure EVE interrupt controller by attaching and interrupt handler to mailbox interrupt (MBOX_INTO). This will ensure that whenever an interrupt occurs in mailbox this interrupt handler gets invoked. Then we enable mail box interrupt in EVE's mailbox module which will raise this interrupt whenever a new message is received by mailbox.

Similarly on DSP side we first setup DSP interrupt controller and and then we enable mailbox interrupt for DSP which will raise an interrupt whenever a new message is received by DSP from EVE. There is a difference between mailbox user id for DSP in case of Vision28 Super and TDA1MEV platforms. In Vision28 Super mailbox user id for DSP is MBOX_USER_1 whereas in case of TDA1MEV it is MBOX_USER_2.

In this example, first EVE writes a message to its mailbox and once DSP receives this message it will acknowledge back to EVE buy writing in mailbox.

Note: While building mailbox example make sure you give proper PLATFORM name for Vision28 Super(vayu) and TDA1MEV (vme)

2.1.2.Mailbox_eve1_to_dsp1_inorder_ack

This example is very similar to the above example. Only difference is that this will continue for 100 iteration. Eve will first write a message mailbox for DSP and DSP will acknowledge back by writing a message to EVE's mailbox. This will continue for 100 iterations.

Note: While building mailbox example make sure you give proper PLATFORM name for Vision28 Super(vayu) and TDA1MEV (vme)

2.1.3.Mailbox_dsp1_all_eves

This example is specifically meant to be run only Vision28 Super (vayu) platform, the reason being this example requires multiple EVE's (in this case 4 EVE's). All the interrupt and mailbox configurations are similar to above examples. From EVE side, each EVE will write into mailbox for DSP. On DSP side, DSP will get interrupt from each EVE and will keep incrementing a count. DSP will wait till it gets interrupt from all the four EVE's. DSP will keep on acknowledging interrupts from EVE mailbox module by writing message back to mailbox. Once it receives interrupt from all the messages it will return PASS value.

Note: While building mailbox example make sure you give proper PLATFORM name for Vision28 Super(vayu)

2.1.4.Eve_bfswtch_arp32_error_intr

The buffer switch controls the ownership of the individual buffers, through the buffer view register and the buffer switch register. The "EVE_MSW_CTL" that controls ownership, and the memory view is implemented by "EVE_MEMMAP". These registers are explained in Section 10.3 of programmers guide. It was discussed that an error in programming the buffer switch, can be detected by use of the "EVE_MSW_ERR_INT".

In this example we enable the "EVE_MSW_ERR_INT" event, which corresponds to the buffer switch error interrupt. In this example, we purposefully set the buffer switch with the incorrect settings and perform an ARP32 data access to a buffer for which ARP32 is not the master. In this example the Bfswitch_error_Intr_Handler" handles the buffer switch error interrupt, In the interrupt service routine we read the registers "EVE_MSW_ERR" which tells us who generated the error, and the address from "EVE_MSW_ERR_ADDR" to which an access was made, when buffer switch setting was incorrect.

2.1.5.Eve_bfswtch_vcop_error_intr

This example is very similar to the above example. In this example again we make use of "EVE_MSW_ERR_INT" interrupt to detect error because of vcop buffer switch. Here we allocate data buffers in "WBUF", "IBUFLA" and "IBUFHA", but purposefully hand off "WBUF", "IBUFFLB" and "IBUFFHB" to VCOP, so that we can trigger a buffer switch error. In interrupt service routine after reading "EVE_MSW_ERR_IRQSTATUS", we write back to it, to clear the interrupt status, so that we can respond to future interrupts.

2.1.6.Program_cache_global_inv

This example shows how to do a global invalidate of program cache present in EVE module. The program cache global invalidate register can be set to invalidate all the lines in the L1P cache by resetting all valid bits. Any CPU fetches put out during the invalidate-all operation will stall until the invalidate-all operation completes, and will subsequently miss in the program cache.

2.1.7.Program_cache_block_inv

The program cache controller supports a programmable invalidate mechanism with which the starting address and the number of words to invalidate can be specified to fire off a range-based invalidation sequence. The programming model for the invalidation mechanism consists of two memory-mapped registers: the Start Address Register that holds the start address and the Byte Count Register that holds the number of bytes to be invalidated.

This example shows how to make use of this feature. The invalidate operation begins immediately on writing into the Byte Count register (max = 0x8000). The application must first set the Start Address Register and then the Byte Count Register to ensure correct operation. The operation itself involves cycling through addresses starting from the Start Address Register value in increments of the cache line size, doing tag lookups to check if the line exists in the cache and if it does, resetting the corresponding valid bit. The Byte Count Register field is reset to zero when the invalidate completes. As the Byte Count Register is readable, a check for zero provides a synchronization event for the application to execute from the region being invalidated or before issuing another range based invalidate. In the course of such an invalidation sequence, any further writes to the Start Address Register or the Byte Count Register is ignored. Note that start the Start Address can be any arbitrary byte address; whereas, the invalidate operation occurs on cache-lines (that is, 32-bit aligned). Thus, the range invalidated is effectively rounded down to the nearest cache-line address relative to the start address, and rounded up to include the entire cache line relative to the end address.

As in the global invalidate case, any CPU program fetches put out during the invalidate-range operation will stall until the invalidation completes. The underlying invalidate operation will begin after any in-flight requests are completed. The single address invalidate feature is mainly for breakpoints set by the debugger, and hence you need not leverage this feature, as it is primarily for the debugger.

2.1.8.Program_cache_software_prefetch

This example shows how to use software directed preload (SDP) capability of program cache module inside EVE subsystem.

Software directed preload (SDP) allows you to request that a range of system memory to be preloaded into the Program Cache. You set the Preload Base Address register, along with a Preload Byte Count (max = 0x8000). Hardware will issue a cache line fill request to the system for the associated line for the programmed address range. Once the data is returned for each line, it is written to the cache (32-B per write, to minimize program stalls) and the tag is marked valid for the new address. This is repeated for every cache line in the requested range. When the operation is complete, the Preload Byte Count register will be read as 0. Software must verify that the previous operation has completed before issuing another SDP operation. Writes to the Preload Base Address or Byte Count while the previous operation is ongoing will be ignored.

Note that the Start Address can be any arbitrary byte address; whereas the preload operation occurs on cache-lines (that is, 32-bit aligned). Thus, the range preloaded is effectively rounded down to the nearest cache-line address relative to the start address, and rounded up to include the entire cache line relative to the end address.

Preload requests operate in parallel with the demand based prefetch block and do not use the buffering provided by the Demand Based Prefetch block. Instead, the return data for Software Directed Preload is written directly to the program cache as highest priority, thus minimizing buffering required and minimizing impact on the system.

2.1.9.Ref_mmu_tlb

This example shows how to program Memory Management Unit (MMU) of eve subsystem. In this example we implement manual TLB programming on the DSP, to setup the basic entries that are needed to run EVE applications. We need to always take up one TLB entry to map the virtual address of 0, which is where ARP32 resets to a valid physical address. The first TLB entry is used to do this, by mapping with the smallest memory size of 4K bytes. The second TLB entry maps the DDR address range with the identity transformation, where we are just saying that for the first 16 Mbytes of DDR, virtual address is the same as physical address. This is the reason why VIRTUAL_ADDRESS2 is the same as PHYSICAL_ADDRESS2. The third TLB region is used to show that if we intend to access say a system address of 4803 0000h and a 1MB region starting at this address. We also take two more TLB addresses to carve up level-3 L3 memory, referred to as on-chip memory OCMCRAM, into two regions of size 4KB and 64KB.

The 4KB page size and 64KB page sizes are normally used for partitioning code segments, while the 1MB and 16MB larger page sizes are used for large data arrays. After setting up the TLB entries, in this example we can access these memories using the virtual address as from EVE side.

You will notice that we have taken the small memory page to map the reset vector. We are leveraging virtual memory address translation on TLB entries 4 and 5, where the virtual address and it's corresponding physical addresses are indeed different. For this example, we choose to map a virtual address of 5030 0000h, maps to a physical address of 4030 0000h. We have also chosen to designate all these TLB entries as mapping little-endian memory regions, with the access width being decided based on CPU access size. Additionally we have decided to lock down these TLB entries from being evicted by setting the preserve bit. This prevents the TLB entry from being flushed, as TLB misses get serviced, allowing to preserve a fast real time response for memory requests that access the regions programmed by these TLB entries. After configuring the required number of TLB entries to carve out a specific virtual memory to physical memory mapping, we enable the MMU hardware. At this point, we can release the ARP32 processor out of reset, by having the DSP write to the appropriate register at the SOC level. The ARP32 processor resets at virtual address 0, which is mapped to 8000 0000h in our example, which is why the reset vector is linked at this address in the EVE software examples.

2.1.10. Ref_mmu_tlb_miss_tlw_dis

This example demonstrates the concept of TLB miss in EVE subsystems MMU.

When EVE makes an access to an address region for which there are no corresponding entries in the MMU, there is a translation look aside buffer miss. This causes EVE to be stalled for a response until the MMU has responded to the TLB miss. The way to catch such events, is to enable the MMU interrupt support, which is routed to the EVE interrupt controller as one of the events. Unlike other interrupt events which can be serviced by ARP32 the MMU interrupt needs to be serviced by an external host most typically an ARM. In order to do this, EVE has up to four output interrupts, which can be routed to external hosts. Please also refer to registers EVE_INTX_OUT_IRQ raw, status, set and clear to figure out which of the 32 events, you want an external host to respond to. The MMU error interrupt is a good example of an interrupt which should be serviced by an external host.

In this example, we use the DSP once again to program the TLB entries. We configure the MMU. We continue to poll on the IRQ raw register to see, if a MMU error interrupt has occurred. On a normal SOC platform, we would have enabled this as an interrupt source. When the MMU error has been detected, we read the fault address and status register to determine why the error was generated.

We program the TLB for the fault address, clear the MMU IRQ status, and return. In a normal SOC platform the function “MMU_Intr_Handler” will run as an interrupt service routine on whichever host is tasked with the job of servicing the MMU error interrupt.

2.1.11. Vcop_done_intr

This example shows the usage of VCOP_DONE interrupt of interrupt controller. In this example , we setup the “vcop_done” event to interrupt “INT4” and enable it. The use of detecting “vcop_done” as an interrupt is fairly limited, as you will be polling because you have sent a “VWDONE” in order to receive an interrupt. This is one of the main reasons that most EVE software examples use polling, as it is possible to finish everything else you can finish within a task of computation, and then poll at the end as opposed to using interrupts, which always has a context switch overhead. The function “vcop_done_intr_Handler” is the interrupt handler for the event “vcop_done”, which is registered in the array “inth_Irq_Handler”. The disabling of the interrupt is done at the end of the test case by calling the “EVE_INTH_InterruptDisable” function, to disable the event.

2.1.12. Vcop_error_intr

The VCOP hardware will report error status due to various error conditions in the VCOP Error register, in addition to generating an interrupt for ARP32 to service. Section 9.3 of EVE’s programmer guide presents the details of various VCOP registers, including the error register. Errors can be triggered due to various conditions such as illegal instructions, incorrectly formed loops, parameter pointer to the loop is not aligned on a 32-bit boundary, parameter register points to out of bound memory region, exceeded maximum iterations as specified in “MAX_ITERS” register, load or store out of bound access from IBUFL, IBUFH, or WBUF, error due to force abort request received.

These error scenarios do not occur during normal code execution. However, when they do occur, it is convenient to receive an interrupt, read and clear the VCOP status register before issuing the next vector core loop for execution. Hence, it is convenient and, in fact, a good recommended practice, to have the vcop error interrupt enabled even during code or application development to catch various errors due to incorrect programmation, as opposed to running incorrect code and the difficulties involved with debugging such code.

This example shows the code for a vector core error interrupt test case, where we purposefully inject this error, by passing the vector core function “my_eve_array_add_uns_char_custom” a parameter pointer which is not 32-bit aligned. The vector core always reads the parameter registers each of which are 16- bits, 16 at a time, taking advantage of the vector core’s 256-bit bandwidth on any given memory. In this example, we map the vector core error interrupt to interrupt 4. We will see in output that interrupt handler “vcop_error_Intr_handler” being executed twice, and the value of the VCOP_ERROR register to be 4, which corresponds to bit 2, of the register. Section 9.3 of EVE’s programming guide shows that this bit is set, if the parameter register is not 32-bit aligned.

2.1.13. Vcop_max_iters_intr

This example shows how to use maximum iteration register. The vector core maximum iterations register is a safety register, by which the user can indicate the maximum number of iterations that no individual loop will exceed. The programmer is expected to analyze his application and program a safe value, which none of the loops are supposed to exceed. This feature is disabled on EVE reset, and the maximum iteration register has a zero value. This feature is intended to catch and stop the vector core as soon as possible, in cases where it receives illegal parameters from memory, and hence gets into a run-away situation. It is possible that without this check, the vector core can run for several iterations, as it supports four nested loops, and each loop count can be a 16-bit value.

In this example we are setting up the interrupt controller to detect interrupts for vector core completion and for vector core error. We have a common interrupt handler function by way of "vcop_done_Intr_Handler", to handle both vector core done, and vector core error events.

Upon seeing a vector core error, all remaining vector core instructions, including instructions of back to back loops will be decoded and flushed, until the vector core sees a "VWDONE" instruction, on which it will send both a vector core done, and a vector core error event, which can raise an interrupt. In our case, since we have both events enabled, we see both a done interrupt, and an error interrupt with the ninth bit, or bit 8 set, which is why vcop_error register has the value of 256. It is the user's responsibility to read and clear the vector core error register. On clearing the error register, we can resume processing, and since we send the same erroneous sequence, we see the interrupt again.

2.1.14. Sctm_vcop_busy_time

Several events of interest can be measured using SCTM. To understand the performance of VCOP on a given loop, we can instrument our code to measure how long VCOP was busy. The vcop_busy SCTM signal indicates the decode time and execution time. The SCTM hardware allows you to drill down into one loop or a sequence of back-to-back loops, and obtain a detailed view of how VCOP or program cache is behaving in the actual hardware and collects this information, without altering the program execution. This example shows how to get vcop busy cycles using SCTM module of EVE subsystem.

2.1.15. sctm_counters_observing_8_events

This example shows how to use multiple counters to observe multiple events simultaneously. We can use all eight counters, and be monitoring multiple events of interest in parallel, as shown in the code. In this example we are monitoring the start of individual VCOP loops, the finishing of the individual VCOP loops (these are pulse events, we get 1 pulse every time VCOP starts or stops), duration VCOP was busy (duration event), number of reads to IBUFL, IBUFH, WBUF, number of writes to IBUFH and WBUF, as the events of interest. We have 2 loops that are called back-to-back, for execution on VCOP, so that the decode of the second loop can happen in the background as the first loop executes.

2.1.16. edma_circ1d_eve

The pointer increments that we can program into the parameter RAM can be either positive or negative, allowing us to implement circular buffers automatically leveraging EDMA without using a CPU to calculate addresses. This example program shows the usage of a parameter RAM set to implement circular buffering for 1D data transfers.

This example uses negative pointer increments to implement circular buffering. We are setting up NUM_TRANSFERS transfers with each transfer implementing a 1D transfer of num_bytes and incrementing the source and destination address (SRCBIDX, DSTBIDX) of num_bytes on even transfers and rewinding the destination and continue incrementing the source (SRCCIDX, DSTCCIDX) by (-

num_bytes) and (num_bytes). This example shows the power of the different indices in the parameter set to automatically calculate addresses.

We need to program the options field to allow it to update source and destination addresses automatically without a CPU programming it by selecting the NORMAL update mode as opposed to STATIC mode for transfers where we use a CPU to calculate a new address prior to every transfer.

2.1.17. edma_circ1d_rewrite_eve

This example is same as previous example except for the fact that in this example EDMA sustains itself by linking to another PaRAM to rewrite the current parameter to restore it.

2.1.18. edma_eve_ROI_w_linking

This example shows how to setup double buffering of ROI blocks via DMA. The input is a greyscale image stored in DDR memory. A set of square ROIs are defined by their <x,y> origins in the arrays ROI_x[] and ROI_y[]. The origin is assumed to be at the top-left corner of the image. Each ROI is transferred from external memory to EVE IBUFL via linked DMA transfers. Prior to submission of the first transfer, the address and dimensions of each ROI block are written to an EDMA PARAM entry. The PARAM of subsequent transfers are referenced in the link address field of the prior ROI block's PARAM. In this example, we have set up circularly linked transfers. For example, if we have defined six ROI blocks, A B C D E and F, after block F has been transferred it will link back to block A for the next transfer. Blocks are double buffered to IBUFL. For example, we can process block C in IBUFLA while transferring block D to IBUFLB.

The ROI block in IBUFL is copied to IBUFH by the vector core. In the output direction, the processed ROI block in IBUFH is transferred to external memory through a series of DMA auto incrementing transfers. These transfers place the processed ROIs contiguously in external memory, per the illustration below.

In this example, we have setup 6 regions of interest, and have submitted 12 block transfers. The circular linking starts the transfer sequence at A again after F has been sent.

ASSUMPTIONS:

Since the vector core kernel processes data per 8 pixels in SIMD, the ROI block width must be a multiple of 8.

2.1.19. edma_eve_double_buffering_alias_view

This example is for setting up double buffering via DMA. The input is a greyscale image, stored in DDR memory, which is divided into 2D blocks that are transferred via DMA for processing in EVE internal memory. The processing step is a simple copy of the block between two EVE internal memory regions. The processed data is then transferred from internal to DDR memory via DMA.

2.1.20. edma_simple_eve

This example is intended to show several simple examples of how to use the register level Chip Support Library (CSL) for EDMA to setup single 1D -> 1D, 1 -> 2D, 2D -> 1D, 2D -> 2D transfers. The aim here is to show a basic template for simple memcpy with the transfer being kicked off by CPU setting up an event. This file also shows a linking with self-chaining example, to demonstrate linked 1D -> 1D and linked 1D -> 2D. It also has an example of auto-increment on 1D -> 1D transfers which posts both intermediate and final transfer completion codes

This page is intentionally left blank

Chapter 5

Frequently Asked Questions

This chapter provides answers to few frequently asked questions related to using this starterware.

3.1. Release Package

Question	Answer
Can this starterware release be used on any eve based platform?	Yes, you can use it. But there are some examples involving multiple EVE's and hence can only be validated on vayu platform
Where are the host (DSP) API's in this release	Host API's are same as the API's for EVE starterware. You just have to link using correct library from the libs folder (Libs folder contain separate libraries for DSP hosts). It is to be noted that not all eve starterware API's are supported from host. For the list of API's supported on DSP kindly refer to section 0
Where are API's to configure DSP interrupts	As DSP interrupts configuration is not part of eve starterware API's, in this release we have moved these API's to examples folder (starterware/examples/common/dsp). Each example currently build these API's as a part of their own build
This is EVE Starterware release why there is a DSP folders inside it	Some of the EVE Starterware API's can also be called from hosts like DSP and hence we have DSP folders which contains DSP specific details

3.2. Code Build and Execution

Question	Answer
Build is not working and not giving any error	Ensure that you have set proper environment variables as described in section 2.3.
Starterware build is not working on windows, giving error "gmake: *** No rule to make target"	Ensure that you have set proper environment variables as described in section 2.3. Make sure that your EVE_SW_ROOT path is not too long. If this path is too long in windows, build can fail. Other way to fix this problem is to create a link to this path. For this you can use subst utility in windows. This will create a new drive with the content of you directory. subst <DRIVE NAME> PATH e.g. subst X: $(EVE_SW_ROOT)
Starterware build not working on windows: Makefile:9: D:workvayueve/modules/starterware/examples/common/common.mk: No such file or directory	Make sure in your environment variable all paths use forward slash "/" instead of backward slash "\". Also do not use any quotes or spaces in the path names. If there are some spaces in path then usethe following way, if we want to set ARP32 compiler path as C:/Program Files/Texas Instruments/ARP32_tools, use set ARP32_TOOLS=C:/PROGRA~1/TEXASI~1/ARP32_tools

Question	Answer
Where can I file gel files needed for running mailbox examples	Kindly look into gel folder present in gels folder located at EVE software root directory. It contains two separate folders for vayu and vme platform.

3.3. Issues with tools

Question	Answer
What tools are required for using this starterware release?	Kindly see section 0 for the list of tools required.