BAM Algorithm Framework User's Guide

Texas Instruments

Last updated on
September 26, 2017

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) Block-based acceleration manager (BAM) Framework on the EVE. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

Intended Audience

This document is intended for system engineers who want to integrate TI's algorithms with other software to build a vision/multimedia system based on the EVE.

This document assumes that you are fluent in the C language, have a good working knowledge of Digital Signal Processing (DSP), digital signal processors, and DSP applications.

How to Use This Manual

This document includes the following chapters:

Chapter 1 - Introduction, provides a brief overview of the contents.

Chapter 2 - Installation Overview

Chapter 3 – Usage of BAM

Chapter 4 - API Reference, refer to doxygen

Chapter 5 – Current limitations, describes the current limitation sof BAM

Chapter 6 - Frequently Asked Questions, provides answers to few frequently asked questions related to using this algorithm.

Chapter 7 – Example code

Chapter 8 - Differences between BAM v2.02 and BAM v2.03

Related Documentation from Texas Instruments

Related Documentation

You can use the following documents to supplement this user guide:

Embedded Vision Engine (EVE) Programmers guide (SPRUHC1A)

EVE Subsystem reference guide (SPRUHF5)

ARP32 CPU and Instruction Set Reference Guide (SPRUHC9)

Embedded Vector Engine (EVE) Programmer's Guide (SPRUHC1A)

VCOP CPU and Instruction Set Reference Guide (SPRUHC0)

VCOP Kernel C Reference Guide (SPRUHB9)

ARP32 Compiler Reference Guide (SPRUH24)

TMS320DM646x DMSoC Enhanced Direct Memory Access (EDMA3) Controller (SPRUEQ5-EDMA3)

Abbreviations

The following are the abbreviations used in this document.

Table 1. List of Abbreviations

Abbreviation	Description
EVE	Embedded Vision Engine

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

When contacting TI for support on this algorithm, quote the product name (Image Resizer Software on EVE) and version number. The version number of the algorithm is included in the Title of the Release Notes that accompanies this algorithm.

Trademarks

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

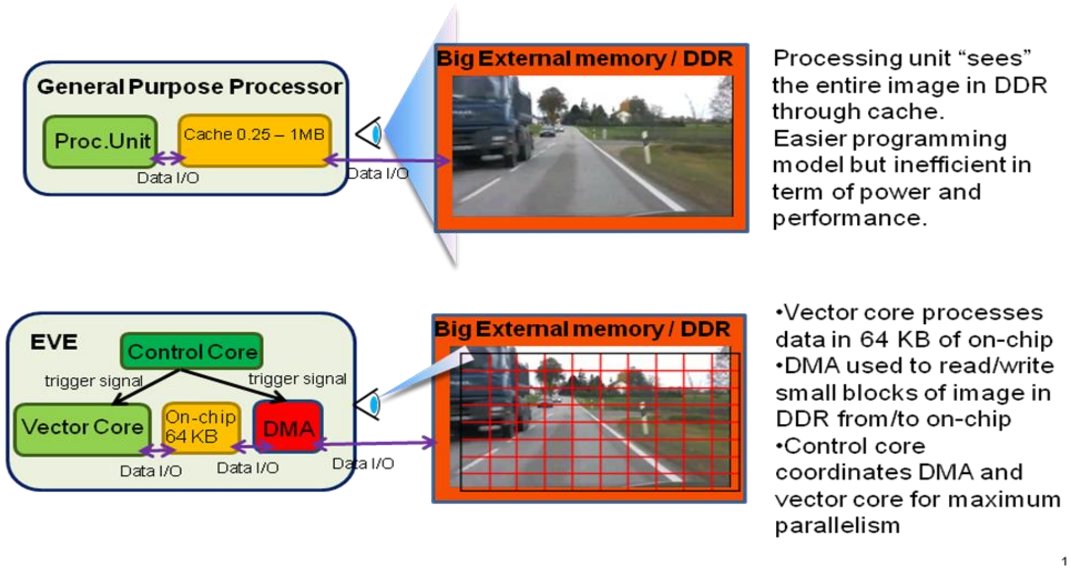
This chapter provides an overview of the BAM algorithm framework.

1.1 Overview of BAM algorithm framework

The goal of the Block-based Acceleration Manager (BAM) is to ease the porting effort of existing image/vision algorithms developed on a PC processor onto TI embedded system by converting the algorithm’s original processing flow into a block-based form that is more embedded processor friendly.

Image/vision processing functions from most PC-based libraries such as openCV can be directly applied to a large data set, typically the size of a QVGA, VGA or HD image frame. However in the embedded processor world, the amount of data that a particular function can efficiently process at once is limited by the size of its on-chip memory, where data to be processed resides. Some embedded general purpose processors (GPP) have cache memory and thus functions will be able to access large data residing in external memory. However image processing algorithms such as filtering require accessing the data in 2-D fashion and this is generally not very well modeled by the cache, resulting in many cache misses, which adversely affect the performance. That’s why the preferred approach is to use DMA to bring the data in a block-based fashion into the on-chip memory for fast processing.

On a processor such as EVE, the processing unit, which is the vector core here, has access to the content of its on-chip memory at any time. But this on-chip memory is only 64KB and cannot hold the entire image at one time. That is why a DMA engine is used to read/write blocks of image data from DDR from/to the on-chip memory. At any time, the vector core can only process the data block present in its on-chip memory. There is a control core that schedules the execution of the vector core and DMA to ensure that every block of the image gets
processed.



Processing unit "sees" the entire image in DDR through cache.
Easier programming model but inefficient in term of power and performance.

•Vector core processes data in 64 KB of on-chip
•DMA used to read/write small blocks of image in DDR from/to on-chip
•Control core coordinates DMA and vector core for maximum parallelism

Figure 1 Differences between GPP and EVE in managing the data

The presence of DMA means an extra hardware module to program and more time spent on implementing the data transport aspect of the algorithm instead of the computation aspect. That is why a framework such as BAM was created, to abstract the DMA data movement so the implementer only focuses on the algorithm computation aspect, which is what he would do if he was developing on GPP or PC.

The way BAM achieves the abstraction of the data movement is by having the programmer describe the processing flow using a DAG (directed acyclic graph) and then use BAM APIs to construct the graph offline or at runtime.
The graph is composed of source/sink node for data input/output as well as processing nodes. Source/sink node are implemented in C code to control the EDMA. Processing nodes are implemented in Kernel-C, the vector core programming language.

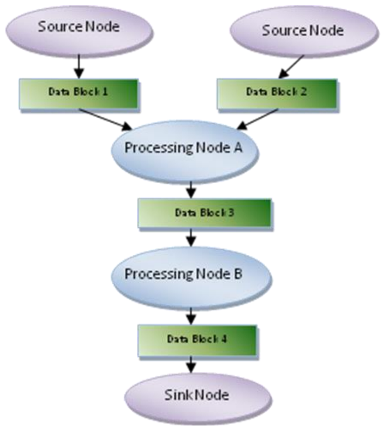


Figure 2 BAM uses direct acyclic graph to abstract the data flow.

To speed up the development time, BAM already comes with implementations of source and sink nodes so the programmer doesn’t need to learn EDMA programming.
BAM also comes with a library of kernels that can be plugged as processing nodes into a graph.

Once constructed, a graph object can be saved offline and be executed at runtime. This separation of graph construction with execution allows reduced execution time of the algorithm since the algorithm can be created statically in an offline manner.

Note that BAM can also be ported onto a GPP as it forces the data to be processed in small 2-D blocks, which is cache friendly.

Other benefits of BAM include:

- Ease the porting from PC to Embedded by providing means to automate the conversion from frame processing to block processing.

- Ease the porting from Embedded to EVE by abstracting the data movement (EDMA in case of EVE) and by automatically handling the on-chip memory allocation.

- Clean method of plugging existing kernels into BAM graph through implementation of helper functions allowing the user to leverage libraries from either TI or 3rd party.

- Misra-C compliant.

1.2 Overview of the package and contents

The package contains the following:

Source code of BAM on EVE

Binary (.lib library) of BAM that can be used on EVE

Implementation of DMA nodes

Chapter 2

Installation Overview

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

2.1 System Requirements

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

2.1.1 Hardware

This module has been built for CentEVE/Vision-Mid-EVE (EVM) and Vayu/Vision28 platforms for both EVE and DSP cores.

2.1.2 Software

This module has been built using ARP32/EVE and C66x CG Tools and tested on CentEVE/Vision-Mid-EVE (EVM) and Vayu EVM/Vision28 Simulator/Virtio/Zebu platforms. Please refer to the user guide of the overall EVE SW package or each DSP algorithm components to know the exact versions of the following tools used for validation.

- Code Composer Studio (CCS)
- EVE Subsystem Simulator (Simpack)
- Code Generation Tools: ARP32

2.2 Installing the Component

The BAM component is released as a windows and linux installer. Run the installer to extract the contents of the package onto your local hard disk. When choosing the destination location, select the path to an existing EVE or DSP software release directory or to update its content with the latest BAM files.

Figure 3 shows the sub-directories that are created by the BAM installation package.

Table 2 provides a description of the sub-directories created. Their contents are source files used to build the BAM algorithm framework library.

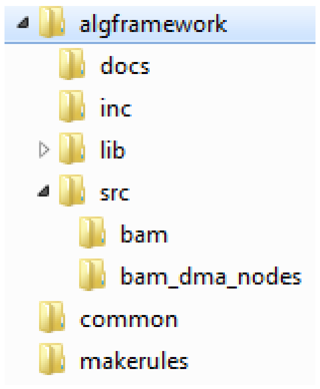


Figure 3 Component Directory Structure

Table 2 Component Directories of BAM release

Sub-Directory	Description
\makerules	Contain build-related configuration files
\common	Contain files that are common to different software components: BAM, DMA_UTILS, algorithms, applets, kernels, etc.
\algframework	Top level folder that contains BAM algorithm framework source files
\algframework\src\bam	Contain sources C and internal header files for creating the BAM lib.
\algframework\src\bam_dma_nodes	Contain sources of DMA nodes implementation
\algframework\inc	Interface of the BAM algorithm framework
\algframework\lib	.lib library of the module is placed in this folder

The EVE software release provides other components that use BAM to demonstrate its capabilities. The table below lists these components not included in the BAM release but included in the EVE software release.

Table 3 Component Directories of EVE sw release used with BAM

Sub-Directory	Description
\algorithms	Contain source files and testbenches of example algorithms that are implemented in form of BAM graphs. They are: • array_op: simple point-to-point addition between 2 frames. • array_op_scalar: simple scalar-to-point addition between a frame and a scalar • filter: apply 5x5 FIR filter to a frame. • ISP: image signal processing pipeline that converts bayer to YUV. • resize: 2x downscaler.

Sub-Directory	Description
\apps	Contain source files and testbenches of productized algorithms that are implemented in form of BAM graphs. They are: • blocks_statistics • fast9_corner_detect • median_filter • imagePyramid_u8 • harrisCornerDetection32 • pyramid_lk_tracker
\kernels\bamdb	Contain the database of the BAM kernels used by the example mentioned above,
\kernels\imgsiglib\Array_op	Contain sources of array_op implementation in kernel-C as well as BAM wrapper files.
\kernels\imgsiglib\Array_op_scalar	Contain sources of array_op_scalar implementation in kernel-C as well as BAM wrapper files.
\kernels\imgsiglib\Filter	Contain sources of filter implementation in kernel-C and natural C as well as BAM wrapper files.
\kernels\kernelslib\vcop_image_bayer2rgb	Contain sources of Bayer to rgb implementation in kernel-C as well as BAM wrapper files. Used in ISP graph.
\kernels\kernelslib\vcop_image_rgb2srgb	Contain sources of rgb to srgb implementation in kernel-C as well as BAM wrapper files. Used in ISP graph.
\kernels\kernelslib\vcop_image_rgb2yuv	Contain sources of rgb to yuv implementation in kernel-C as well as BAM wrapper files. Used in ISP graph.
\kernels\kernelslib\vcop_deinterleave	Contain sources of deinterleave implementation in kernel-C as well as BAM wrapper files. Used in resizer graph.
\kernels\kernelslib\vcop_gammacorrection	Contain sources of gamma correction implementation in kernel-C and natural C as well as BAM wrapper files. Used in ISP graph.
\kernels\kernelslib\vcop_interleave	Contain sources of interleave implementation in kernel-C as well as BAM wrapper files. Used in resizer graph.
\kernels\kernelslib\vcop_subsampling	Contain sources of subsampling implementation in kernel-C as well as BAM wrapper files. Used in ISP graph.
\kernels\kernelslib\vcop_whitebalance	Contain sources of white balance implementation in kernel-C and natural C as well as BAM wrapper files. Used in ISP graph.

We recommend the reader to look at the implementation of the imagePyramid_u8 applet which is a good template for any applet implemented on top of BAM.

2.3 Before Building

Install the *DMA_UTILS* package since there is some header file dependency. Make sure that the directory *dmautils* is installed at the same level as *algframework*.

Open the file makerules\config.mk and edit the following environment variables: DSP_TOOLS, ARP32_TOOLS, UTILS_PATH to point to the correct locations.

The makefile allows to build a PC version of the library which is used for host emulation. Indeed all the EVE or DSP algorithms can be built to run on PC since each underlying component such as DMA_UTILS, BAM can be build in host emulation mode using Visual C++ as compiler.

2.4 Building the package

The package comes with pre-built binaries for EVE and DSP and there is no need to build them again. Hower, in case if there is a need or if the PC version is required, this section explains the procedure.

Open a windows' command window and go to the algframework folder and type one of the gmake command, depending on the desired target core and build configuration:

	Release	Debug
DSP	`gmake CORE=dsp`	`gmake TARGET_BUILD=debug CORE=dsp`
EVE	`gmake CORE=eve`	`gmake TARGET_BUILD=debug CORE=eve`
PC host emulation for DSP	`gmake CORE=dsp TARGET_PLATFORM=PC`	`gmake CORE=dsp TARGET_PLATFORM=PC TARGET_BUILD=debug`
PC host emulation for EVE	`gmake CORE=eve TARGET_PLATFORM=PC`	`gmake CORE=eve TARGET_PLATFORM=PC TARGET_BUILD=debug`

The libraries are generated in *algframework/lib/dsp*, *algframework/lib/eve* or *algframework/lib/PC/dsp, algframework/lib/PC/eve* . To clean the build, insert the word 'clean' after 'gmake', ex: `gmake clean TARGET_BUILD=debug CORE=dsp` .

If you get an error `".\inc\starterware.h", line 22: fatal error: could not open source file "csl_edma.h"`, then it means the DMA_UTILS package was not installed.

Note that, for successful build of the PC version, Microsoft Visual C++ must have been installed.

2.5 Configuration Files

None.

2.6 Uninstalling the Component

To uninstall the component, delete the directory from your hard disk.

Chapter 3

Usage of BAM

This chapter provides a detailed description of the different steps required to implement an algorithm using BAM.

There are 3 stages involved in the usage of BAM.

1. Graph Design stage

In this is a paper and pencil activity, the programmer models the complete data flow in the form of a directed acyclic graph, composed of nodes and edges.

2. Graph Implementation stage
   a) First implement each processing kernel associated to the nodes that will compose the graph.
   b) The programmer then uses `BAM_createGraph()` to create a graph object. Multiple graphs can be created and their respective objecter container can be saved for subsequent execution.

3. Graph Execution stage

The programmer invokes `BAM_process()` to execute the algorithm represented by the graph handle passed to the function. Multiplegraphs can be executed sequentially by passing a different handler each time BAM_process() is called. Context save and restore of the internal state of the on-chip memories, EDMA registers, processor status is handled automatically when switching from one graph to another.

3.1 Graph Design

The type of graph supported by BAM is the directed acyclic graph like the one in Figure 4. In other words, each node represents a processing step and each edge of the graph represents a data connection. Usage of arrow to represent the edges allows the differentiation between two types of data connection: incoming arrows to a node represent input data whereas outgoing arrows from a node represent output data. The graph is directed because the data always flows in one direction and each node's processing is started in the order specified by the arrows. No loopback within the graph is allowed, since it is not possible for a node to read data that will be produced by a downstream node in the future.

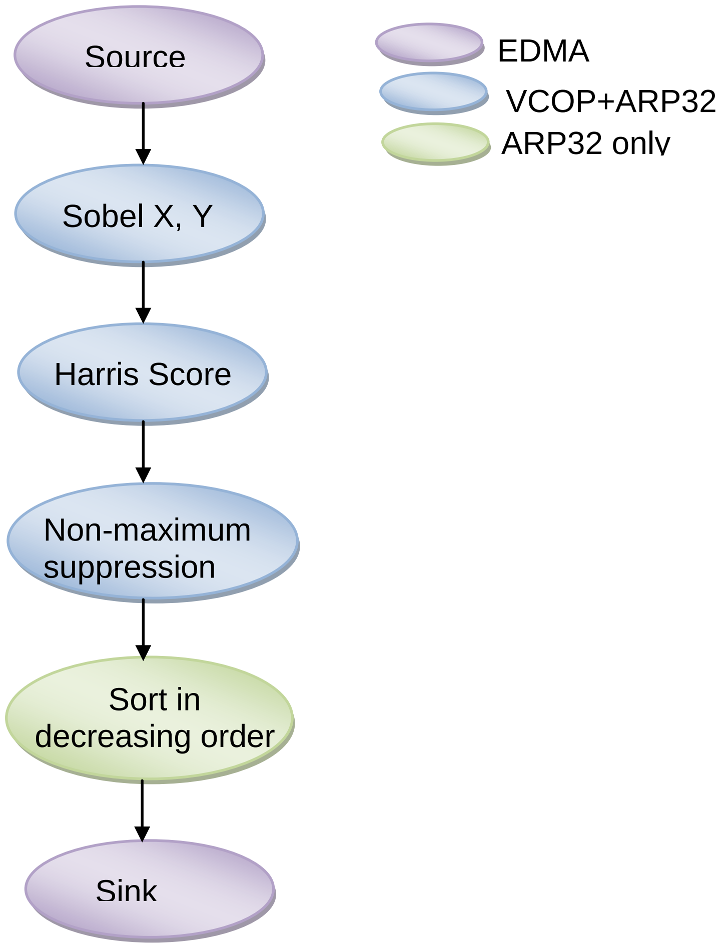


Figure 4 Example of directed acyclic graph

There are three different types of nodes:

A processing or compute node reads input data from on-chip memory, processes it and writes it back to on-chip memory. In Figure 4, the Sobel (X,Y), Harris Score, Non-maximum suppression, Sort in decreaser order are all processing nodes. A processing node can have one or more input/output edges. For any node ‘n’, we call upstream node, a node that is connected to ‘n’ by an input edge and downstream node, a node that that is connected to ‘n’ by an output edge.

A source node is a special node that reads data from external memory at a block granulariy and stores it in on-chip memory.

A sink node is a special node that writes data present in on-chip memory to external memory.

When executing the graph, BAM will run all the source and sink nodes in parallel with the processing nodes in order to hide the data transfers with the computation. This is highly effective when data transfers are implemented using EDMA.

Current implementation of BAM allows up to 256 nodes in a graph and only **one source node** and **one sink node**. If the graph requires several inputs our outputs then the single source or sink node must be able to handle multiple I/O. The default source and sink nodes that TI supplies can handle up to 10 inputs and outputs. One output of an upstream node can be connected to many inputs of a downstream node while the reverse is not true.

EVE case: a compute node can either be implemented in a mix of C and Kernel-C language or in C only. In the former case, the compute node will use both ARP32 and VCOP hardware processing unit and in the later case, the compute node will use ARP32 only.

3.2 Graph Implementation

3.2.1 Node's kernel implementation

Any node graph performs some processing that is implemented either in C or Kernel-C language. Typically, the processing could be implemented by just one function, which is called a compute kernel. Outside of BAM context, an algorithm programmer can just call this compute kernel to apply the desired processing on a block of data located in on-chip memory, provided that input pointers, output pointers, function arguments are correctly initialized. In the case of BAM, the higher level of abstraction requires more inter-operability and interaction between the kernel and the framework, that is why extra functions must be implemented to achieve this.

There are two sets of functions that must be implemented:

Helper functions: These functions are called by BAM when the application calls `BAM_createGraph()`. Indeed nodes are connectet together based on the graph description provided by the caller and each node's helper functions provide to BAM all the necessary memory requirements.

Execute functions: these are functions called whenever `BAM_process()` is called by the application. They include the typical compute kernel but also some initialization functions that are called at the beginning of every frame or the very first time the instance of the graph is executed. Each kernel can also provide a control function, which is called by `BAM_controlNode()` to get or set specific kernel's parameters such as threshold values, coefficient values, DMA transfer parameters after a graph is created and before each `BAM_process()` call. This allows dynamic control of the graph execution at every frame boundary.

In addition a data structure of type `BAM_KernelInfo` associated to the kernel must be declared and initialized.

The next pagraph exposes this structure and the following paragraphs provide details on each set of functions.

Note they are several example files that the reader can use to assist him/her in the understanding of the different concepts exposed hereinafter. These files correspond to existing kernels whose implementation has been ported to BAM:

Kernel	**Graph**	**Files location**
Point-to-point array operation	array op applet	*kernels\imgsiglib\Array_op\bam_helper*
Point-to-point scalar to array	array scalar op applet	*kernels\imgsiglib\Array_op_scalar\bam_helper*

operation		
Filter operation	filter applet	*kernels\imgsiglib\Filter\bam_helper*
Gamma correction	ISP applet	*kernels\kernelslib\vcop_image_gammaCorrection\bam_helper*
RGB to YUV color conversion	ISP applet	*kernels\kernelslib\vcop_image_rgb2yuv\bam_helper*
RGB to sRGB color conversion	ISP applet	*kernels\kernelslib\vcop_image_rgb2srgb\bam_helper*
Bayer to RGB color conversion	ISP applet	*kernels\kernelslib\vcop_image_bayer2rgb\bam_helper*
2x subsampling	ISP applet	*kernels\kernelslib\vcop_image_subSampling\bam_helper*
white balance	ISP applet	*kernels\kernelslib\vcop_image_whiteBalance\bam_helper*
de-interleave	Resizer applet	*kernels\kernelslib\vcop_image_deinterleave\bam_helper*
interleave	Resizer applet	*kernels\kernelslib\vcop_image_interleave\bam_helper*
pyramid on luma	Resizer applet	*kernels\kernelslib\vcop_image_pyramid_u8\bam_helper*
pyramid on chroma	Resizer applet	*kernels\kernelslib\vcop_image_pyramid_uv_u8\bam_helper*
2x2 block averaging	5-levels Image pyramid	*kernels\imgsiglib\BlockAverage2x2\bam_helper*

Table 4 List of kernels ported to BAM

3.2.1.1 Structure `BAM_KernelInfo`

This structure provides to BAM contextual information about a kernel:

Members of BAM_KernelInfo

BAM_KernelId	**kernelId**
	System-wide unique identifier of a kernel. Initially set to 0 when the structure is defined in bam_<kernel_name>_helper_func.c Later it gets automatically initialized by**BAM_initKernelDB()** with the ID defined in the kernel database passed to the function **BAM_initKernelDB()**.
uint32_t	**kernelContextSize**
	Size of the kernel's context structure BAM_<kernel_name>_Context, which is a private structure specific to the kernel that contains members such as pointers to the input, internal scratch, output data blocks as well as arguments for the compute kernel. This structure content must follow a specific pattern, described in the BAM user's guide. By convention, the structure is declared in file bam_<kernel_name>_helper_func.h and defined in file bam_<kernel_name>_helper_func.c The operator sizeof() should be used to

initialize this member kernelContextSize for ease of code maintenance. Ex: sizeof(BAM_<kernel_name>_Context). Sink node can have kernelContextSize set to 0. This in turn indicates to BAM to use the source node's context as sink node's context as well. This feature is useful when source and sink nodes uses the same underlying hardware (EDMA) to implement the data transfers. Since the context serves as a proxy to the hardware resource, when source and sink nodes use the same hardware, it is more efficient that they share the same context. So setting kernelContextSize to 0 for a sink node is a way to indicate to BAM that this sink node uses the same context as the source node of the graph.

uint32_t	**kernelArgSize**
	Size of the kernels' arguments structure BAM_<kernel_name>_Args that is exposed to the application. Such structure becomes the list of parameters that will be passed to the kernel's compute function. Since it is exposed to the application, it must be declared in the kernel public header file which is bam_<kernel_name>.h . The way the application will communicate the argument's values is by providing pointers to initialized BAM_<kernel_name>_Args structures through the nodes' list passed to **BAM_createGraph()**. The operator sizeof() should be used to initialize this member kernelArgSize for ease of code maintenance. Ex: sizeof(BAM_<kernel_name>_Args).

BAM_CoreType	**coreType**
	Type of the processing core on which the kernel is implemented on: BAM_EVE_ARP32, BAM_EVE_VCOP, BAM_EVE_DSP_C64x, etc.

uint8_t	**nodeType**
	Type of node the kernel must be associated to: BAM_NODE_COMPUTE, BAM_NODE_SOURCE, BAM_NODE_SINK. More...

uint8_t	**numInputDataBlocks**
	Number of input data blocks to the kernel.

uint8_t	**numOutputDataBlocks**
	Number of output data blocks to the kernel.

uint8_t	**numRecInternal**
	Number of on-chip memory records required by the kernel during kernel execution. These memory segments are internal to the kernel and are used as scratch buffers or to store constant.

The convention is to define this structure in file

kernels\kernelslib\<kernel_name>\bam_helper \bam_<kernel_name>_helper_func.c

where <*kernel_name*> is the name of the computation associated to the kernel such as: sobel, harris_score, etc.

An example for the point-to-point array operation kernel is:

```
BAM_KernelInfo gBAM_TI_arrayOpKernel =
{
  0, /*kernelId */
  sizeof(BAM_Image_arrayOp_Context),  /*kernelContextSize*/
  sizeof(BAM_Image_arrayOp_Args),     /*kernelArgSize */
  BAM_EVE,                            /*coreType*/
  BAM_NODE_COMPUTE,                   /*nodeType */
  NUM_IN_BLOCKS,                      /*2*/
  NUM_OUT_BLOCKS,                     /*1 */
  NUM_INTERNAL_BLOCKS                 /*1 */
};
```

The macro symbols `NUM_IN_BLOCKS,  NUM_OUT_BLOCKS, NUM_INTERNAL_BLOCKS` should be defined in the file *kernels\kernelslib\<kernel_name>\bam_helper \bam_<kernel_name>_helper_func.h.*

For the example of point-to-point array operation kernel, the function adds two data block together to produce one output block, thus two input blocks and one output block are required.

EVE case:

If the kernel is implemented using a mix of C language and kernel-C language, the member `BAM_KernelInfo::coreType` **must be set to either** `BAM_EVE` **or** `BAM_EVE_VCOP` **since both ARP32 and VCOP hardware resources will be utilized. When VCOP is involved,** **`NUM_INTERNAL_BLOCKS`** **is set to at least 1 because an internal block must be allocated to store the param registers. The ARP32 code must not access the image buffers, only VCOP code can. If there is a need for the ARP32 code to access the image buffer then the portion of the code must be separated out and spun into a new kernel in which** `BAM_KernelInfo::coreType` **is set to** `BAM_EVE_ARP32`**.**

Any kernel implemented using C code only (no kernel-C) that accesses the image buffers must have member `BAM_KernelInfo::coreType` **set to** `BAM_EVE_ARP32`**. BAM does take care of the buffer switching so ARP32 has ownership of the image buffers when it is its turn to access them. The kernel implementer does not need to call** `VCOP_BUF_SWITCH_SET()` **inside the compute function.**

This file must also contain the kernel's context structure:

```
typedef struct _bam_<kernel_name>_Context
{
/* Must always follow this order: pInternalBlock[], pInBlock[],
OutputBlock[], args */
    void *pInternalBlock[NUM_INTERNAL_BLOCKS];
    void *pInBlock[NUM_IN_BLOCKS];
    void *pOutBlock[NUM_OUT_BLOCKS];
    BAM_<kernel_name>_Args kernelArgs;
} BAM_<kernel_name>_Context;
```

As mentioned in the comments, the kernel's context structure must always start with an array of pointers to the internal memory blocks, followed by an aray of pointers to input and output blocks and finally the kernel argument's structure. If any member is missing or the order is different then the graph creation or graph execution will fail or misbehave.

The member `kernelArgs` is of type `BAM_<kernel_name>_Args`, which is a custom structure. The implementer of the kernel has total freedom on the definition of this structure. Here is the declaration for the array op kernel example:

```
typedef struct _arrayOp_bam_image_arrayOp_args
{
  unsigned int inputA_blk_width;   /* width(stride) of input block A */
  unsigned int inputA_blk_height;  /* height of input block A */
  unsigned int inputB_blk_width;   /* width(stride) of input block B */
  unsigned int inputB_blk_height;   /* height of input block B */
  unsigned int output_blk_width;    /* width(stride) of output block */
  unsigned int output_blk_height;   /* height of output block */
  unsigned int compute_blk_width; /* number of elements processed per row*/
  unsigned int compute_blk_height;  /* number of row processed */
  unsigned int rnd_shift;           /* number of bits to right shift before
rounding */
}BAM_Image_arrayOp_Args;
```

The declaration of the structure must be exposed to the application and it is recommended to put it in the public header file *bam_<kernel_name>.h.*

For sink node's kernels that are in charge of data transfers, the member `kernelContextSize` can be set to 0 to indicate to BAM to use the same context as the source node.

3.2.1.2 Helper functions

The helper functions are grouped in the structure `BAM_KernelHelperFuncDef` made up of 2 functions:

```
typedef struct {
 /* Helper functions are defined on the core on which the constructor
   is executed: Arp32, M3, M4, A15.
 */
BAM_KernelGetMemRecFunc getMemRec; /* Function to get kernel's memory
records. BAM will call it when BAM_createGraph() is invoked by the applet and
will use information such as size, alignment, memory space to automatically
allocate the physical memory block for each record. */
BAM_KernelSetMemRecFunc setMemRec; /* Function used by the kernel to copy the
pointer of each memory records back to the context structure. The location of
the physical memory blocks were determined by BAM's memory allocated and this
function allows the location information to be communicated back to the
kernel.*/
} BAM_KernelHelperFuncDef;
```

This structure definition must be present in the *bam_<kernel_name>_helper_func.c file* along with the implementation of its member functions `BAM_KernelHelperFuncDef::getMemRec()` and `BAM_KernelHelperFuncDef::setMemRec()`.

(i) getMemRec

This function is used by BAM to query a kernel's memory records. BAM calls it whenever the application executes `BAM_createGraph()`. BAM uses the information obtained from `BAM_KernelHelperFuncDef::getMemRec()` to automatically allocate the memories in an optimal way.

The prototype is:

```
typedef BAM_Status (*BAM_KernelGetMemRecFunc)(
        const void *kernelArgs,
        BAM_MemRec *memRecInternal,
        BAM_MemRec *memRecOutputDataBlock,
        uint8_t *numRecInternal,
        uint8_t *numRecOutputDataBlock);
```

The member function `BAM_KernelHelperFuncDef::getMemRec()` must return two arrays of memory records associated to the kernel:

- `memRecInternal`: list the internal memory blocks required by the kernel.
- `memRecOutputDataBlock`: list the output memory blocks required by the kernel.

Note the absence of memory record for the input blocks. Indeed, during the graph creation process, since nodes are connected to each other via edges, input blocks from a node are also output blocks from an upstream node and thus their memory records do not need to be specified in order to avoid redundancy.

The memory records are of type `BAM_MemRec` declared in *algframework/inc/bam_types.h*:

Members of BAM_MemRec

uint32_t	**size**
	size in bytes of this structure
int32_t	**alignment**
	Alignment of the block's start address in memory.
BAM_MemSpace	**space**
	Automatically filled by BAM, location where the block must be allocated: VCOP image buffer, working memories, ARP32 data memory or external memory.
BAM_MemAttrs	**attrs**
	Specifies BAM memory attributes.
void *	**base**
	Can be automatically filled by BAM if static memory allocation disabled , point to the allocated block.

void *	**ext**
	For internal usage: extra pointer used for context save/restore of persistent internal memory.

char *	**name**
	Automatically filled by BAM, pointer to the string of the node to which this memory record belongs, for easy tracing during debugging.

The type `BAM_MemSpace` is an `int32_t`.

EVE case: **`BAM_MemSpace`** can have the following values: **`BAM_MEMSPACE_IBUFLA`**, **`BAM_MEMSPACE_IBUFHA`**, **`BAM_MEMSPACE_IBUFLB`** **`BAM_MEMSPACE_IBUFHB`**, **`BAM_MEMSPACE_WBUF`**, **`BAM_MEMSPACE_ANY_VCOPBUF`**, **`BAM_MEMSPACE_EXTMEM`**.

and `BAM_MemAttrs` is defined as follow:

BAM_MemAttrs

uint8_t	**nodeIndex**
	For internal BAM usage, Index of the node in the graph.

uint8_t	**memType**
	For internal BAM usage, Type of the memory: BAM_MEMTYPE_NODEMEM_INPUT, BAM_MEMTYPE_NODEMEM_OUTPUT, BAM_MEMTYPE_NODEMEM_INTERNAL.

uint8_t	**dataBlockIndex**
	For internal BAM usage, Corresponding data block index for backward reference.

uint8_t	**memAttrs**
	Attribute of the memory: BAM_MEMATTRS_SCRATCH: Memory space with this attribute CAN be re-used by other nodes to allow overlay and ARE NOT saved & restored during context switching. BAM_MEMATTRS_PERSIST: Memory space with this attribute CANNOT be re-used by other nodes and ARE saved & restored during context switching. BAM_MEMATTRS_CONST: Memory space with this attribute CANNOT be re-used by other nodes and ARE NOT saved & restored during context switching.

Fortunately, BAM already prefills most of the member when it calls `BAM_KernelHelperFuncDef::getMemRec()` and only the size member needs to be filled.

For instance, the implementation of `getMemRec()` for array op kernel is:

```
static BAM_Status BAM_Image_arrayOp_getMemRecFunc
(
    const void *kernelArgs,
    BAM_MemRec *internalBlock,
    BAM_MemRec *outBlock,
    uint8_t    *numInternalBlocks,
    uint8_t    *numOutBlocks
)
{
    BAM_Image_arrayOp_Args *args = (BAM_Image_arrayOp_Args *) kernelArgs;

    internalBlock[PARAMS_IDX].size =
2*vcop_array_ADD_uchar_uchar_param_count();
    outBlock[0].size = (args->output_blk_width * args->output_blk_height);

    return 0;
}
```

EVE case: The internal block that is reserved to store the param registers must have its size set to 2*vcop_<kernel_name>_param_count();

Note that in the example above, the output block's size is computed using the kernel's arguments `args->output_blk_width` and `args->output_blk_height`. Since BAM calls each node's kernel `BAM_KernelHelperFuncDef::getMemRec()` function during the execution of `BAM_createGraph()`, the application will have already initialized `args->output_blk_width` and `args->output_blk_height` through the list of nodes passed to `BAM_createGraph()`. Please refer to `BAM_createGraph()` for details. Since `getMemRec()` can access the argument structure, it can also check the validity of the values. For instance `args->output_blk_width` must usually be aligned with the SIMD width of the processor (16 in the case of EVE) and an `assert(args->output_blk_width % 16== 0)` directive could be added to the implementation of `getMemRec()` .

The default member values in `internalBlock[]` and `outputBlock[]` prefilled by BAM upon calling `getMemRec()` are listed in table

Member's name	Default values
size	0
alignment	32 bytes in case of EVE (to match alignment of vcop_malloc() used by BAM).

space	In case of EVE
	`internalBlock[].space= BAM_MEMSPACE_WBUF`
	`outputBlock[].space= BAM_MEMSPACE_ANY_VCOPBUF`

attrs	`internalBlock[].attrs.memAttrs= BAM_MEMATTRS_CONST`
	`outputBlock[].attrs.memAttrs= BAM_MEMATTRS_SCRATCH`
	Other members of `attrs` are undefined.

base	NULL
ext	NULL
name	NULL

Table 5 Default values for internalBlock[] and outputBlock[]

Note that BAM automatically presets all output blocks's memory attribute to `BAM_MEMATTRS_SCRATCH`. This corresponds to the majority of use case, in which the output data produced by a node is consumed by the downstream node and its memory area can be reused after consumption. BAM built-in memory allocator can figure out the lifetime of any `BAM_MEMATTRS_SCRATCH` memory for reuse by other nodes.

EVE case: when a memory attribute is set to `BAM_MEMATTRS_SCRATCH`, the associated memory space should be set to `BAM_MEMSPACE_ANY_VCOPBUF`. This allows BAM to automatically figure out the best location of the memory block, to minimize I/O conflict when accessing multiple pieces of data in VCOP's memory. That's why the space of the output memory blocks is defaulted to `BAM_MEMSPACE_ANY_VCOPBUF`. However `getMemRec()` can still override the settings to a specific location such as `BAM_MEMSPACE_IBUFLA` in case it results in faster memory access .

Likewise, BAM automatically presets all internal blocks' memory attribute to `BAM_MEMATTRS_CONST`. This corresponds to many use cases, in which the purpose of the internal data is to store constants that cannot be over written at any time during the processing. Such data corresponds to filter coefficients, table lookup, etc. Note that for these constant values to be saved and restored during graph context switching, `BAM_MEMATTRS_PERSIST` should be used instead so the content of the block is saved to external memory and can be restored whenever the original graph is executed again.

There are also many times when an internal memory block must be allocated as scratch. This happens when the computation kernel produces several intermediary results before arriving to the output. These intermediary results do not need to remain in memory after the kernel's computation finishes and that is why the attribute `BAM_MEMATTRS_SCRATCH` should be used. In such case `BAM_KernelHelperFuncDef::getMemRec()` must overwrite the corresponding members `attrs.memAttrs` of the internal blocks used as scratch with `BAM_MEMATTRS_SCRATCH`.

EVE case: when a memory attribute is set to `BAM_MEMATTRS_CONST` or `BAM_MEMATTRS_PERSIST`, the associated memory space should be set to `BAM_MEMSPACE_WBUF`. That's why the space of the internal memory blocks is defaulted to `BAM_MEMSPACE_WBUF` because of their default attribute which is `BAM_MEMATTRS_CONST`.

Lastly the prototype of `getMemRec()`:

```
typedef BAM_Status (*BAM_KernelGetMemRecFunc)(
      const void *kernelArgs,
      BAM_MemRec *memRecInternal,
      BAM_MemRec *memRecOutputDataBlock,
      uint8_t *numRecInternal,
      uint8_t *numRecOutputDataBlock);
```

shows two arguments referenced by pointers: `numRecInternal` and `numRecOutputDataBlock`. Before calling `getMemRec()`, BAM actually presets *`numRecInternal` and *`numRecOutputDataBlock` with the values `numRecInternal` and `numOutputDataBlocks` taken from the BAM_KernelInfo structure. So most of the time `getMemRec()` does not need to alter their values.

The only time when `BAM_KernelHelperFuncDef::getMemRec()` should modify *`numRecInternal` or *`numRecOutputDataBlock` is when the actual number of memory records initialized is less than the initial value. Indeed, sometimes the number of internal or output blocks is dependent on the kernel's arguments, which are known at runtime only. It is then convenient to specify a worst case number for the members `numRecInternal` and `numOutputDataBlocks` of

`BAM_KernelInfo`. The real values are later derived from the argument structure passed to `BAM_KernelHelperFuncDef::getMemRec()`, during runtime.

Indeed in case of EDMA source node, the number of output blocks must correspond to the number of transfer channels actually used. By default, at implementation time, `BAM_KerneInfo .numRecOutputDataBlock= NUM_OUT_BLOCKS= 10` which enables up to 10 EDMA channels to read 10 data frames. At runtime, the application sets the effective number of channels to member `numChannels` of the argument structure `DMANODE_EdmaInterfaceArgs`. Upon the application calling `BAM_createGraph()`, the EDMA kernel function `getMemRec()` overrides *`numRecOutputDataBlock` with the runtime value. Please refer to file *algframework\src\bam_dma_nodes\Eve_edma.c*, function `DMANODE_edmaGetMemRecFunc()` for implementation details.

After BAM uses the information obtained from `BAM_KernelHelperFuncDef::getMemRec()` and automatically allocates the memories, BAM needs to feed the memory records' pointers value back to the kernel. Indeed the compute kernel needs this information in order to process the data at the correct location. This information must be stored in the kernel's context structure that we discussed in page 4.

This feedback process can be done either automatically by BAM if the member function `BAM_KernelHelperFuncDef::setMemRec()` of the structure `BAM_KernelHelperFuncDef` is set to NULL. If the supplied function if not NULL, then BAM executes it instead of automatically handling the feedback process. All example kernels use the automatic way, except for the EDMA kernels. When the automatic way is used, it is imperative that the context structure follows this convention: the structure must always start with an array of pointers to the internal memory blocks, followed by an array of pointers to input and output blocks and finally the kernel argument's structure.If any member is missing or the order is different then the graph creation or graph execution will fail or misbehave.
BAM also copies the argument values set by the application to the context structure.

(ii) setMemRec

```
typedef BAM_Status (*BAM_KernelSetMemRecFunc)(
       const BAM_MemRec *memRecInternal[],
       const BAM_MemRec *memRecInputDataBlock[],
       const BAM_MemRec *memRecOutputDataBlock[],
       uint8_t numRecInternal,
       uint8_t numRecInputDataBlock,
       uint8_t numRecOutputDataBlock,
       void *kernelContext, const void *kernelArgs);
```

When member `setMemRec` of `BAM_KernelHelperFuncDef` is not NULL then this function must perform the following:

Copy each element in the array `memRecInternal[]` to the array `kernelContext->pInternalBlock[]`.

Copy each element in the array `memRecInputDataBlock[]` to the array `kernelContext->pInBlock[]`.

Copy each element in the array `memRecOutputDataBlock[]` to the array `kernelContext->pOutBlock[]`.

Copy the content of the structure `kernelArgs` to `kernelContext->kernelArgs`.

Here is the example code for the array op kernel :

```
static BAM_Status BAM_Image_arrayOp_setMemRecFunc
(
  const BAM_MemRec *internalBlock[],
  const BAM_MemRec *inBlock[],
  const BAM_MemRec *outBlock[],
  uint8_t           numInternalBlocks,
  uint8_t           numInBlocks,
  uint8_t           numOutBlocks,
  void             *kernelContext,
  void             *kernelArgs
)
{
  BAM_Image_arrayOp_Context *context = (BAM_Image_arrayOp_Context *)
kernelContext;
  BAM_Image_arrayOp_Args *args = (BAM_Image_arrayOp_Args *) kernelArgs;

  if (numInternalBlocks != NUM_INTERNAL_BLOCKS)
  {
    PRINTF("ERROR: Unexpected numInternalBlocks recieved
SetMemRecFunc()!!!");
  }
  if (numInBlocks != NUM_IN_BLOCKS)
  {
    PRINTF("ERROR: Unexpected numInBlocks recieved SetMemRecFunc()!!!");
  }
  if (numOutBlocks != NUM_OUT_BLOCKS)
  {
    PRINTF("ERROR: Unexpected numOutBlocks recieved in SetMemRecFunc()!!!");
  }

  context->kernelArgs=  *args;

  context->pInBlock[0] = inBlock[0]->base;
  context->pInBlock[1] = inBlock[1]->base;
  context->pOutBlock[0] = outBlock[0]->base;
  context->pInternalBlock[PARAMS_IDX] = internalBlock[PARAMS_IDX]->base;

  return BAM_S_SUCCESS;
}
```

As a reminder, the context structure is a custom structure specific to the kernel, which must follow this template:

```
typedef struct _bam_<kernel_name>_Context
{
/* Must always follow this order: pInternalBlock[], pInBlock[],
OutputBlock[], args */
    void *pInternalBlock[NUM_INTERNAL_BLOCKS];
    void *pInBlock[NUM_IN_BLOCKS];
    void *pOutBlock[NUM_OUT_BLOCKS];
    BAM_<kernel_name>_Args kernelArgs;
} BAM_<kernel_name>_Context;
```

When `setMemRec` of `BAM_KernelHelperFuncDef` is NULL then compliance of the context structure with the pattern above allows BAM to automatically initialize it. This saves the implementer of the kernel from implementing the function. That's why in the example code for array op, `BAM_Image_arrayOp_setMemRecFunc` is actually commented out in the source file and the associated `BAM_KernelHelperFuncDef` is defined as follow:

```
BAM_KernelHelperFuncDef gBAM_TI_arrayOpHelperFunc =
{
  &BAM_Image_arrayOp_getMemRecFunc,
  NULL
};
```

While the `getMemRec()` and `setMemRec()` helper functions are exectued during graph creation, the next set of function called execute functions are executed during graph's execution.

3.2.1.3 Execute functions

The execute functions are grouped in the structure `BAM_KernelExecFuncDef` made up of 6 functions:

```
typedef struct
{
    BAM_KernelFunc kernelInitInstance;  /**< One time initialization function
of the kernel. Called the first time the graph is execture or after a context
restore */
    BAM_KernelFunc kernelInitFrame;     /**< Initialization function of the
kernel which should be called before every frame processing */
    BAM_KernelFunc kernelCompute;       /**< Kernel function which performs
the computation on every block of data */
    BAM_KernelFunc kernelWait;          /**< Kernel Function which waits for
the completion of computations performed, only for source and sink node */
    BAM_KernelCtlFunc kernelCtl;        /**< Control function of the kernel
called by BAM_controlNode() */
    BAM_KernelCustom kernelCustom;      /**< Custom function of the kernel*/
} BAM_KernelExecFuncDef;
```

This structure definition must be present in the *bam_<kernel_name>_exec_func.c file* along with the implementation of its member functions.

Not every implementation of these member functions need to be provided. If a member function is a NULL handle then BAM does not execute it.

EVE case: At minimum, the `kernelInitInstance` **and** `kernelCustom` **members must be provided. The implementation of** `kernelInitInstance()` **simply calls** `vcop_<kernel_name>_init()` **to initialize the param registers whereas** `kernelCustom` **should be assigned the pointer to** `vcop_<kernel_name>_vloops()`**. The** `kernelCompute` **does not need to be implemented because when it is** `NULL`**, BAM automatically invokes the** `kernelCustom` **function (which is assigned** `vcop_<kernel_name>_vloops()`**). This is to avoid the unnecessary task to implement** `kernelCompute` **as a wrapper around** `vcop_<kernel_name>_vloops()`**. In the situation where many** `vcop_<kernel_name>_vloops()` **need to be called sequentially or conditionnally then the implementation for such wrapper is necessary. See for example the filter kernel example in:** *kernels\imgsiglib\Filter\bam_helper\bam_image_filter_exec_funcs.c,* **which has a different vloop per type of elements processed.**

3.2.2 Kernel database

All the kernels that will be used in the graph nodes must be regrouped into a kernel database.

Actually two database arrays must be defined:

- An array of elements `BAM_KernelHostDBdef`:

```
typedef struct {
    BAM_KernelInfo *kernelInfo; /**< Pointer to the kernel's contextual
information structure BAM_KernelInfo */
    BAM_KernelHelperFuncDef *kernelHelperFunc; /**< Pointer to the structure
BAM_KernelHelperFuncDef that list the helper functions */
    char *name;    /**< Unique name of the kernel */
    BAM_KernelId kernelId; /**< Unique ID of the kernel */
} BAM_KernelHostDBdef;
```

  The member `name` is used for debugging. Note that the memory record structure `BAM_MemRec` also has a member `name`. BAM actually copies the kernel's name from the database to all the memory records associated to the kernel after `BAM_KernelHelperFuncDef::getMemRec()` is called. We will see later that `BAM_createGraph()` actually returns all the allocated memory records to the application, allowing for inspection and easy tracability of which memory record belongs to which kernel, by looking up the name member.

  The `kernelID` is of type `BAM_KernelID` which is a `uint32_t` and is an arbitrary number that identifies the kernel and must be unique within the database.

  Examples of a database array of elements `BAM_KernelHostDBdef` can be found in each applet's file *apps/<applet>/algo/src/<applet>_graph.c* . There is also a master database located in *kernels/bam_kernel_database.c* but it is no longer maintained as the preferred solution is to define local database for each applet, in order to limit codesize. The example below shows the master database.

```
static BAM_KernelHostDBdef bamKernelHostDB[] = {
  { &gBAM_TI_dmaReadKernel, &gBAM_TI_dmaReadKernelHelperFunc,  "ti_dma_read",
      BAM_TI_KERNELID_DMAREAD },
 { &gBAM_TI_dmaWriteKernel, &gBAM_TI_dmaWriteKernelHelperFunc,
      "ti_dma_write", BAM_TI_KERNELID_DMAREAD },
 { &gBAM_TI_downsampleu8Kernel, &gBAM_TI_downsampleu8HelperFunc,
      "ti_image_pyramid_u8_one_level", BAM_TI_KERNELID_PYRAMID_DOWN_U8 },
      . . .
      . . .
 { &gBAM_TI_arrayOp_scalar_Kernel, &gBAM_TI_arrayOp_scalar_HelperFunc,
      "ti_image_arrayOp_scalar", BAM_TI_KERNELID_ARRAY_OP_SCALAR }
};
```

  The ID `BAM_TI_KERNELID_DMAREAD` , `BAM_TI_KERNELID_DMAREAD` , `BAM_TI_KERNELID_PYRAMID_DOWN_U8` , etc, are defined in the file *kernels\bamdb\bam_kernel_database.h* . There must be a one-to-one mapping between the kernels and the IDs and the database must list the kernels in sequentially increasing value of kernelID. The function `BAM_initKernelDB()` will actually check this condition and return `BAM_E_DATABASE_KERNELID` if it is not met.

- An array of elements `BAM_KernelExecFuncDBdef`:

```
typedef struct
{
    BAM_KernelInfo *kernelInfo;  /**< Pointer to the kernel's contextual
information structure BAM_KernelInfo */
    BAM_KernelExecFuncDef *kernelExecFunc;  /**< Pointer to the structure
BAM_KernelExecFuncDef that list the execution functions  */
    char *name;     /**< Unique string denoting kernel name*/
    BAM_KernelId kernelId;  /**< Unique ID of the kernel*/
} BAM_KernelExecFuncDBdef;
```

An example of such array can be found in the master database file *kernels\bamdb\bam_kernel_database.c*

```
static BAM_KernelExecFuncDBdef bamKernelExecFuncDB[] =
{
{ &gBAM_TI_dmaReadKernel, &gBAM_TI_dmaReadKernelExecFunc, "ti_dma_read",
BAM_TI_KERNELID_DMAREAD },
{ &gBAM_TI_dmaWriteKernel, &gBAM_TI_dmaWriteKernelExecFunc, "ti_dma_write",
BAM_TI_KERNELID_DMAWRITE },
{ &gBAM_TI_downsampleu8Kernel, &gBAM_TI_downsampleu8ExecFunc,
"ti_image_pyramid_u8_one_level", BAM_TI_KERNELID_PYRAMID_DOWN_U8 },
    . . .
    . . .
{ &gBAM_TI_arrayOp_scalar_Kernel, &gBAM_TI_arrayOp_scalar_ExecFunc,
"ti_image_arrayOp_scalar", BAM_TI_KERNELID_ARRAY_OP_SCALAR }
};
```

Furthermore, these two arrays are grouped into a single database structure of type `BAM_KernelDBdef`:

```
typedef struct
{
    const int32_t numBAMKernels; /**< Number of kernels  recorded in the
database */
    const BAM_KernelHostDBdef *kernelHostFuncDB; /**< Point to the
BAM_KernelHostDBdef database */
    const BAM_KernelExecFuncDBdef *kernelExecFuncDB; /**< Point to the
BAM_KernelExecFuncDBdef database */
} BAM_KernelDBdef;
```

This is illustrated in the example file *kernels\bamdb\bam_kernel_database.c:*

```
BAM_KernelDBdef gBAM_TI_kernelDBdef =
{
    sizeof(bamKernelExecFuncDB) / sizeof(bamKernelExecFuncDB[0]),
    bamKernelHostDB,
    bamKernelExecFuncDB
};
```

Reference to this structure can be passed to functions `BAM_initKernelDB()` and to member kernelDB of `BAM_CreateGraphParams` used by `BAM_createGraph()`.

This database concept is useful for trying out different implementations of the same kernel. Imagine we have two implementations of Harris Corner, one is by vendor A and the other one is by vendor B. The user can just modify the database to choose which implementation it wishes to use. The Harris Corner kernel ID would retain the same value but the value of `kernelInfo, kernelHelperFunc, kernelExecFunc` would change.
Another usage is during the development stage, one could have a database for natural C implementation and one database for optimized kernel C implementation.

At runtime, before a kernel database can be used for the graph creation process, it must be initialized by calling `BAM_initKernelDB()`.

```
BAM_Status BAM_initKernelDB(BAM_KernelDBdef *kernelDBDef);
```

3.2.3 Graph creation at runtime

Once the graph topology is designed and each kernel's helper and exec functions are implemented, the programmer can create the graph by calling the function `BAM_createGraph()`. This creation process is done at runtime and returns a graph object that can be statically linked. (The feature of linking static graph object has not been validated in the present release).

To illustrate the concepts explained hereinafter, the reader can refer to the implementation file *algorithms\src\isp\src\isp_graph.c*, which implements the creation of graph corresponding to an ISP processing chain as shown in Figure 5.

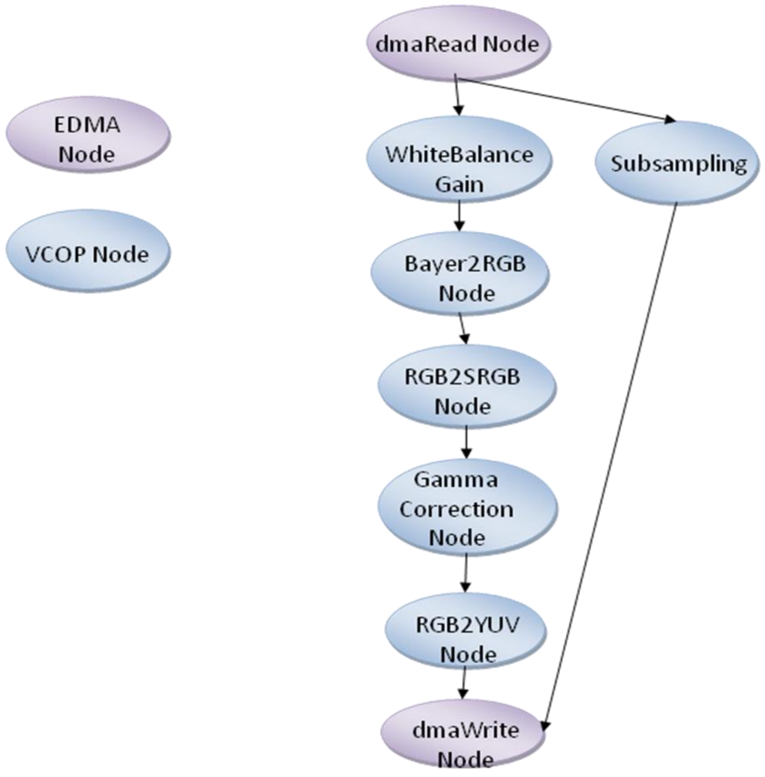


Figure 5 Graph of an ISP processing chain

3.2.3.1 BAM_createGraph()

```
BAM_Status BAM_createGraph(BAM_CreateGraphParams *createParams,
BAM_GraphHandle *pGraphHandle)
```

Create the graph using the description pointed by createParams of type `BAM_CreateGraphParams`, which contains a list of the nodes with their arguments as well as a list to the edges.

Members of BAM_CreateGraphParams

BAM_CoreType	**coreType**
	Type of the processing core on which the graph will execute.
BAM_KernelDBdef *	**kernelDB**
	Point to the database of processing kernels that the graph nodes will use. The database must have been previously initialized using**BAM_initKernelDB()**.
BAM_NodeParams *	**nodeList**
	Point to the list of parameters for the nodes that make up the graph.
BAM_EdgeParams *	**edgeList**
	Point to the list of parameters for the edges that makes up the graph.
void *	**graphMem**
	Point to on chip memory in which BAM will store the graph object, for EVE, it is recommended to be in DMEM.
uint32_t	**graphMemSize**
	Size in bytes of the block of memory pointed by graphMem.
uint32_t	**graphMemConsumed**
	Upon returning from **BAM_createGraph()** the actual number of bytes consumed in graphMem is available here, if BAM_creategraph() returns BAM_E_BLOCK_DIM_TOO_BIG then check if it is because graphMemSizeConsumed > graphMemSize.
void *	**onChipScratchMem**
	Point to on chip scratch memory in which BAM will store the graph object, for EVE, it is recommended to be DMEM.
uint32_t	**onChipScratchMemSize**
	Size in bytes of the block of memory pointed by onChipScratchMem.

uint32_t	**onChipScratchMemConsumed**
	Upon returning from **BAM_createGraph()** the actual number of bytes consumed in onChipScratchMem is available here, if BAM_creategraph() returns BAM_E_BLOCK_DIM_TOO_BIG then check if it is because onChipScratchMemConsumed > onChipScratchMemSize.
void *	**extMem**
	Point to external memory (DDR usually) that will be used to store context saving restore data.
uint32_t	**extMemSize**
	Size in bytes of the block of memory pointed by extMem.
uint32_t	**extMemConsumed**
	Upon returning from **BAM_createGraph()** the actual number of bytes consumed in extMem is available here , if BAM_creategraph() returns BAM_E_OUT_OF_MEM then check if it is because extMemConsumed > extMemSize.
BOOL	**useSmartMemAlloc**
	1: memory allocation is done dynamically and list of memory records is available in memRec after **BAM_createGraph()** returns. Location of the list will be in onChipScratchMem so application should save it before recycling onChipScratchMem for other purpose 0: memory allocation is done statically by using the list of memory records pointed by memRec.
BAM_MemRec *	**memRec**
	Point to list of memory records. When useSmartMemAlloc=true, it is produced by **BAM_createGraph()**, when useSmartMemAlloc=false, caller must initialize it to a list of existing memory records.
uint32_t	**numMemRec**
	When useSmartMemAlloc== true, number of memory records returned by **BAM_createGraph()**. When useSmartMemAlloc=false, this member is not relevant.

BOOL	**optimizeBlockDim**
	If optimizeBlockDim== true, **BAM_createGraph()** will search for the optimium block dimensions and return them in blockDimParams, note that useSmartMemAlloc must be set to true otherwise BAM_E_UNSUPPORTED is returned.
BAM_InitKernelsArgsFunc	**initKernelsArgsFunc**
	Pointer to user-defined callback function invoked by **BAM_createGraph()** to initialize all the kernel arguments present in nodeList When optimizeBlockDim== false, it is called only one time. When optimizeBlockDim== true, it is called more than one time.
void *	**initKernelsArgsParams**
	Point to the custom argument structure passed to the user defined callback function **initKernelsArgsFunc()**
BAM_BlockDimParams	**blockDimParams**
	Pointer to this sctructure is passed to the callback function **initKernelsArgsFunc()** and specifies the recommended processing block width and height. Note that **initKernelsArgsFunc()** can modify these values by incremeting or decrementing by blockWidthStep or blockHeightStep in order to meet some constraints. The final values picked up **initKernelsArgsFunc()** are returned into blockDimParams for the caller to use. When optimizeBlockDim== true,**BAM_createGraph()** modifies the values in blockDimParams in search for the optimal ones.

The two most important members are `BAM_CreateGraphParams::nodeList` and `BAM_CreateGraphParams::edgeList` as they carry the description of the graph.

`BAM_CreateGraphParams::nodeList` is an array of definitions of nodes composing the graph. Each definition is of type `BAM_NodeParams`:

BAM_NodeParams

uint8_t	**nodeIndex**
	index of the node in the graph. Arbitrarily defined by the application.
BAM_KernelId	**kernelId**

	Unique ID of the kernel associated to the node, must correspond to an ID in the kernel database pointed by kernelDB from BAM_CreateGraphParams.
void *	**kernelArgs**
	Pointer to the kernel's arguments structure with initialized values.

The list must always end with a DUMMY entry that has `BAM_NodeParams::nodeIndex` set to `BAM_END_NODE_MARKER= 255` (defined in *bam_common.h*). If omitted, BAM will scan the list beyond the last node and incorrectly build the graph.

The member nodeIndex must correspond to the index of the node in the array `BAM_CreateGraphParams::nodeList.` The order of the nodes in the array also dictates the order in which they are executed. For instance, a node with nodeIndex=0 is the first node listed in the array and also the first node being executed. Most of of the time, such a node is a source node. A node with nodeIndex=1 is the second node listed in the array and the second being executed, etc.

`BAM_NodeParams::kernelId` must match the ID of the kernel associated to the node. The value in `BAM_NodeParams::kernelId` must be listed in the database pointed by `BAM_CreateGraphParams::kernelDB`.

Here is an example of nodes list for the case of ISP graph:

```
#define SOURCE_NODE                       0
#define SUBSAMPLING_NODE                  1
#define WHITEBALANCE_GAINS_NODE           2
#define BAYER2RGB_NODE                    3
#define RGB2SRGB_NODE                     4
#define GAMMACORRECTION_NODE              5
#define RGB2YUV_NODE                      6
#define SINK_NODE                         7

static const BAM_NodeParams NODELIST[]= {
 {SOURCE_NODE, BAM_TI_KERNELID_DMAREAD,(void*)&dmaReadKernelArgs},
 {SUBSAMPLING_NODE, BAM_TI_KERNELID_SUBSAMPLING, (void*)&sdmaReadKernelArgs},
 {WHITEBALANCE_GAINS_NODE, BAM_TI_KERNELID_APPLY_WHITEBALANCE_GAINS,(void*)&whiteBalancekernelArgs},
 {BAYER2RGB_NODE, BAM_TI_KERNELID_BAYER2RGB, (void*)&bayer2rgbkernelArgs},
 {RGB2SRGB_NODE, BAM_TI_KERNELID_RGB2SRGB, (void*)&rgb2srgbkernelArgs},
 {GAMMACORRECTION_NODE, BAM_TI_KERNELID_GAMMACORRECTION, (void*)&gammaCorrectionkernelArgs},
 {RGB2YUV_NODE,BAM_TI_KERNELID_RGB2YUV,(void*)&rgb2yuvkernelArgs},
 {SINK_NODE,BAM_TI_KERNELID_DMAWRITE,(void*)&dmaWriteKernelArgs},
 {BAM_END_NODE_MARKER, 0, NULL},
 };
```

Note that each member `BAM_NodeParams::kernelArgs` must point to the kernel's argument structure. In the example above, these members are set to point to `dmaReadKernelArgs, dmaReadKernelArgs,whiteBalancekernelArgs, bayer2rgbkernelArgs, rgb2srgbkernelArgs, gammaCorrectionkernelArgs, rgb2yuvkernelArgs, dmaWriteKernelArg` which are declared as global variables defined at compile time. However these kernel argument structures could have also been instantiated at runtime and this case the initialization of the handlers `BAM_NodeParams::kernelArgs` would have to be done at runtime.

The initialization of the contents of these kernel argument structures is done by the callback function `BAM_CreateGraphParams::initKernelsArgsFunc().` It is the responsibility of the kernel integrator/graph creator to implement this function, which interface must follow this template:

```
typedef BAM_Status (*BAM_InitKernelsArgsFunc)(void *customArg,
BAM_BlockDimParams *blockDimParams);
```

An example implementation of this function can be found in Chapter 7. Having this function as a callback is part of a mechanism that finds the optimum processing block's dimensions, explained in the next paragraph 3.2.3.2 .

`BAM_CreateGraphParams::edgeList` is an array of definitions of all the edges interconnecting the nodes of the graph. Each definition is of type `BAM_EdgeParams`:

BAM_EdgeParams

struct {	
uint8_t **id**	
	Id of the upstream node.
uint8_t **port**	
	Data port of the upstream node, this edge connects to.
}	**upStreamNode**
	Upstream node id and port the edge connects to.
struct {	
uint8_t **id**	
	Id of the downstream node.
uint8_t **port**	
	Data port of the downstream node, this edge connects to.
}	**downStreamNode**
	Downstream node id and port the edge connects to.

An edge connects an upstream node to a downstream node. Since a node can have several inputs/outputs, the connection point, called 'port', must be specified. The port is an index value defined in the kernel's public header file *kernels\kernelslib\<kernel_name>\bam_<kernel_name>.h* .

The list must always end with a DUMMY entry that has `id` set to `BAM_END_NODE_MARKER= 255` (defined in *bam_common.h*) for both upstream and downstream node. If omitted, BAM will scan the list beyond the last edge and incorrectly build the graph.

Here is an example of edges list for the case of ISP graph:

```
static const BAM_EdgeParams EDGELIST[]= {\
{{SOURCE_NODE, BAM_SOURCE_NODE_PORT1}{WHITEBALANCE_GAINS_NODE, BAM_IMAGE_WB_INPUT_PORT}},
{{SOURCE_NODE, BAM_SOURCE_NODE_PORT1},{SUBSAMPLING_NODE, BAM_IMAGE_SUB_SAMPLING_INPUT_PORT}},
{{SUBSAMPLING_NODE, BAM_IMAGE_SUB_SAMPLING_OUTPUT_PORT},{SINK_NODE, BAM_SINK_NODE_PORT2}},
{{WHITEBALANCE_GAINS_NODE, BAM_IMAGE_WB_OUTPUT_PORT},{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_INPUT_PORT}},
{{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_R},{RGB2SRGB_NODE,BAM_IMAGE_RGB2sRGB_INPUT_PORT_R}},
{{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_G},{RGB2SRGB_NODE,BAM_IMAGE_RGB2sRGB_INPUT_PORT_G}},
{{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_B},{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_INPUT_PORT_B}},
{{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_R},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_R}},
{{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_G},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_G}},
{{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_B},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_B}},
{{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_R},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_R}},
{{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_G},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_G}},
{{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_B},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_B}},
{{RGB2YUV_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_YUV},{SINK_NODE, BAM_SINK_NODE_PORT1}},
{{BAM_END_NODE_MARKER, 0},{BAM_END_NODE_MARKER, 0}},
};
```

Important note: If one single port of an upstream node has more than one connection, which happens in case of a fork in the graph, then all these connections must be clustered together. For instance in the above ISP graph, the port `BAM_SOURCE_NODE_PORT1` of the `SOURCE_NODE` has two connections: one to `WHITEBALANCE_GAINS_NODE` and one to `SUBSAMPLING_NODE`. These connections must appear one afte the other in edge list. If there is a connection involving another node or another port between them then graph creation would be incorrect.

Thus the following edge list would be incorrect because the two connections of the source node are separated by the subsampling node:

```
        static const BAM_EdgeParams EDGELIST[]= {\
        {{SOURCE_NODE, BAM_SOURCE_NODE_PORT1}{WHITEBALANCE_GAINS_NODE, BAM_IMAGE_WB_INPUT_PORT}},
        {{SUBSAMPLING_NODE, BAM_IMAGE_SUB_SAMPLING_OUTPUT_PORT},{SINK_NODE, BAM_SINK_NODE_PORT2}},
        {{SOURCE_NODE, BAM_SOURCE_NODE_PORT1},{SUBSAMPLING_NODE, BAM_IMAGE_SUB_SAMPLING_INPUT_PORT}},
        {{WHITEBALANCE_GAINS_NODE, BAM_IMAGE_WB_OUTPUT_PORT},{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_INPUT_PORT}},
        {{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_R},{RGB2SRGB_NODE,BAM_IMAGE_RGB2sRGB_INPUT_PORT_R}},
        {{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_G},{RGB2SRGB_NODE,BAM_IMAGE_RGB2sRGB_INPUT_PORT_G}},
        {{BAYER2RGB_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_B},{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_INPUT_PORT_B}},
        {{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_R},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_R}},
        {{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_G},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_G}},
        {{RGB2SRGB_NODE, BAM_IMAGE_RGB2sRGB_OUTPUT_PORT_B},{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_INPUT_PORT_B}},
        {{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_R},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_R}},
        {{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_G},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_G}},
        {{GAMMACORRECTION_NODE, BAM_IMAGE_GAMMA_CORRECTION_OUTPUT_PORT_B},{RGB2YUV_NODE, BAM_IMAGE_RGB2YUV_INPUT_PORT_B}},
        {{RGB2YUV_NODE, BAM_IMAGE_BAYER2RGB_OUTPUT_PORT_YUV},{SINK_NODE, BAM_SINK_NODE_PORT1}},
        {{BAM_END_NODE_MARKER, 0},{BAM_END_NODE_MARKER, 0}},
        };
```

Please refer to **Error! Reference source not found.** for another example of invalid edge list.

Every output port of a node must have at least one connection. The number of output ports is equal to the value provided by the member `BAM_KernelInfo::numOutputDataBlocks` of the kernel associated to the node. BAM_createGraph() will check for consistencies between the edge list and the number of output data blocks declared by each kernel, and will return the error `BAM_E_PORT_NOT_CONNECTED` if it finds any unconnected port.

There may be cases where the output from a particular node's port is not consumed. In this case, the port must be connected to the NULL node, which is identified by `BAM_NULL_NODE = 254.`

The remaining members of `BAM_CreateGraphParams`: `graphMem, graphMemSize, graphMemConsumed, onChipScratchMem, onChipScratchMemSize, onChipScratchMemConsumed, extMem, extMemSize, extMemConsumed` are used to manage the amount of memory required by the function `BAM_createGraph()` for its internal operation. Note that these members are not used by BAM to determine the memory requirement for the graph processing. We have already seen that memory requirement for the processing are communicated separately through the helper function `BAM_KernelHelperFuncDef::getMemRec()` of each kernel and are internally allocated by BAM as they target processing blocks that are passed along the graph.

Memory requirements for BAM's internal operation relates to the amount of memory used to build the graph, to store the graph context into external memory, etc. Basically it is the overhead memory due to the presence of a framework. The allocation of this type of memory is not handled internally by BAM but by the application. There are 3 such memories:

1. Memory pointed by the member `graphMem`, into which `BAM_createGraph()` will store the graph object after successful execution. Preferabily, `graphMem` should be located in on chip memory (DMEM in case of EVE), for faster execution of `BAM_createGraph()`. Also, the returned value in the handler `pGraphHandle` will be set to `graphMem`. Note that the member `graphMemsize` must be initialized with the size of the allocated memory as input argument. The reason is that `BAM_createGraph()` will check this value against the true memory requirement of the graph and if it is too small, it returns the error satus `BAM_E_INVALID_MEM_SIZE`. In addition to returning an error, it also sets the member `graphMemConsumed` to the size in bytes that `graphMemsize` should be at least equal to. The programmer must then update the code accordingly by allocating a larger memory and recompile it. Note that `BAM_createGraph()` could still return `BAM_E_INVALID_MEM_SIZE` even after the adjustment. If it fails again, the programmer should reinspect the new value `graphMemConsumed` and use it to allocate larger a memory one more time. The reason for this iterative behavior is that `BAM_createGraph()` is not able to derive the exact amount of memory consumed until it finishes to build the graph completely. Up to 3 iterations of code execution, updating and rebuilding might be necessary to get to the real `graphMemsize` . A strategy to avoid this, is to allocate some 'big' amount of memory (around 16 KB is considered big for a graph object) to ensure the function will return `BAM_S_SUCCESS` . Actually when `graphMemsize` is large enough, not only status `BAM_S_SUCCESS` is returned but also `graphMemConsumed` is set with the actual number of bytes consumed by the graph object. This is useful in case the application has sized `graphMemConsumed` with too much margin and thus his returned value gives the programmer the opportunity to readjust the memory more optimally.

   This way of managing memory requirement is unusual but is a good compromise in the context of a system that cannot use dynamic memory allocation due adherence to safety standard such as MISRA-C. In such a system, a framework cannot allocate memory by its own and the application must do it. Usually there are functions provided by the framework that the application can call to query the memory requirements in advance. However we didn't choose this approach because the implementation of these query functions would be inefficient as they would execute many steps of `BAM_createGraph`(). Since anyway the allocation must be static, the implemented approach actually works well. It might just take a couple of iterations of code execution, updating and rebuilding, to arrive to the result.

2. Memory pointed by the member `onChipScratchMem`, which `BAM_createGraph()` uses as a scratch memory for the duration of its execution. This scratch memory can be released after `BAM_createGraph()` returns and can be recycled for other purpose. The mechanism behind the determination of the optimal memory size `onChipScratchMemSize`, using feedback from the returned status and the value in `onChipScratchMemConsumed` is the same as for the previous case of `graphMem`.

3. Memory pointed by the member `extMem`, which `BAM_createGraph()` uses to save the content of kernel's persistent data during graph context saving . This memory should be located off-chip as the size will typically exceed on-chip memory. The mechanism behind the determination of the optimal memory size `extMemSize`, using feedback from the returned status and the value in `extMemConsumed` is the same as for the previous case of `graphMem` and `onChipScratchMem`.

We previously stated that the allocation of the memory used for processing is handled internally by BAM. Since BAM's memory allocation relies partly on vcop_malloc(), a dynamic allocator that has its heap in VCOP memory, this would cause issue if one wants to strictly adhere to a MISRA-C rule that forbids usage of malloc(), no matter where the heap is located.

To address this issue, `BAM_createGraph()` offers the option to disable the dynamic allocation and to accept a static memory partition defined at compile time.

Indeed when the member `useSmartMemAlloc=1`, `BAM_createGraph()` will take the array of memory records pointed by member `memRec` in order to partition the memory. Now the remaining question is how does one generate a static array of memory records ? The answer is by first running `BAM_createGraph()` in dynamic allocation mode by setting `useSmartMemAlloc` to 0 and saving the resulting array pointed by `memRec`. The member `numMemRec` returns how many memory records have been generated so it is possible to write the array into a file that will be later included in the build process. Only at production stage the programmer needs to set `useSmartMemAlloc`=1 and use the static array.

During development stage, when `useSmartMemAlloc=0,` quiet often the dynamic allocation will fail due to the values used to initialize some kernel arguments related to processing block width and height. Indeed there is a proportional relationship between the dimensions of the processing block and the size of the internal or output memory's record that is specified by the `BAM_KernelHelperFuncDef::getMemRec()` helper function. The more nodes a graph has, the more constrained each kernel's processing block's width and height become as more memory blocks have to share the same amount of on-chip memory. When `BAM_createGraph()` fails due to shortage of on-chip memory for all the processing blocks, it returns `BAM_E_BOCK_DIM_TOO_BIG` and the programmer must decrease the processing block dimensions. He doesn't want to decrease them by too much in order not to loose performance since the larger the block is, the less overhead there is. The search for the optimum block dimensions by this error and trial method can become very tedious, even more so that during development stage, there are often modifications made to the graph. An addition or deletion of a node alters the memory partition and one must resume the search process every time it happens.

Fortunately if `optimizeBlockDim==1`, the function `BAM_createGraph()` automates such process which would be tedious if done programmatically.

3.2.3.2 Block dimensions optimization

This feature is enabled when `BAM_CreateGraphParams::optimizeBlockDim==1` and need the following members of `BAM_CreateGraphParams` to be initialized to work:

BAM_InitKernelsArgsFunc	**initKernelsArgsFunc**
	Pointer to user-defined callback function invoked by **BAM_createGraph()** to initialize all the kernel arguments present in nodeList When optimizeBlockDim== false, it is called only one time. When optimizeBlockDim== true, it is called more than one time.
void *	**initKernelsArgsParams**
	Point to the custom argument structure passed to the user defined callback function **initKernelsArgsFunc()**
BAM_BlockDimParams	**blockDimParams**
	Pointer to this sctructure is passed to the callback function **initKernelsArgsFunc()** and specifies the recommended processing block width and height. Note that **initKernelsArgsFunc()** can modify these values by incremeting or decrementing by blockWidthStep or blockHeightStep in order to meet some constraints. The final values picked

	up **initKernelsArgsFunc()** are returned into blockDimParams for the caller to use. When optimizeBlockDim== true,**BAM_createGraph()** modifies the values in blockDimParams in search for the optimal ones.

These flow diagrams describe the innerworking of the block’s dimensions optimization process which is broken into 3 steps:

- Step #1: find optimum block width
- Step #2: find optimum block height
- Step #3: Use optimum block width and block height to set final memory partition

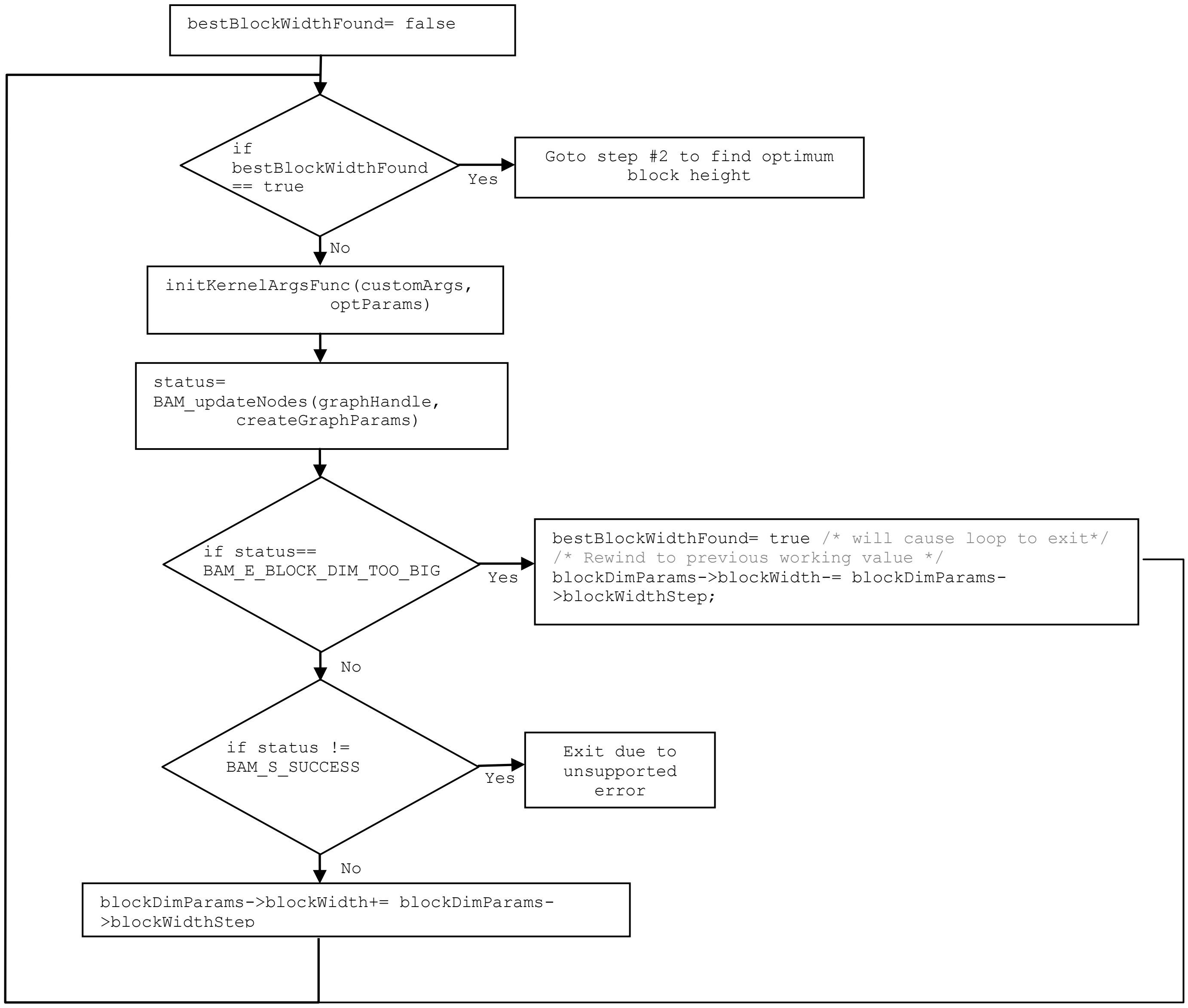

bestBlockWidthFound= false
if bestBlockWidthFound == true
Yes
Goto step #2 to find optimum block height
No
initKernelArgsFunc(customArgs, optParams)
status= BAM_updateNodes(graphHandle, createGraphParams)
if status== BAM_E_BLOCK_DIM_TOO_BIG
Yes
bestBlockWidthFound= true /* will cause loop to exit*/
/* Rewind to previous working value */
blockDimParams->blockWidth-= blockDimParams->blockWidthStep;
No
if status != BAM_S_SUCCESS
Yes
Exit due to unsupported error
No
blockDimParams->blockWidth+= blockDimParams->blockWidthStep

Figure 6 Flow diagram for step #1 of the block's dimensions optimization process

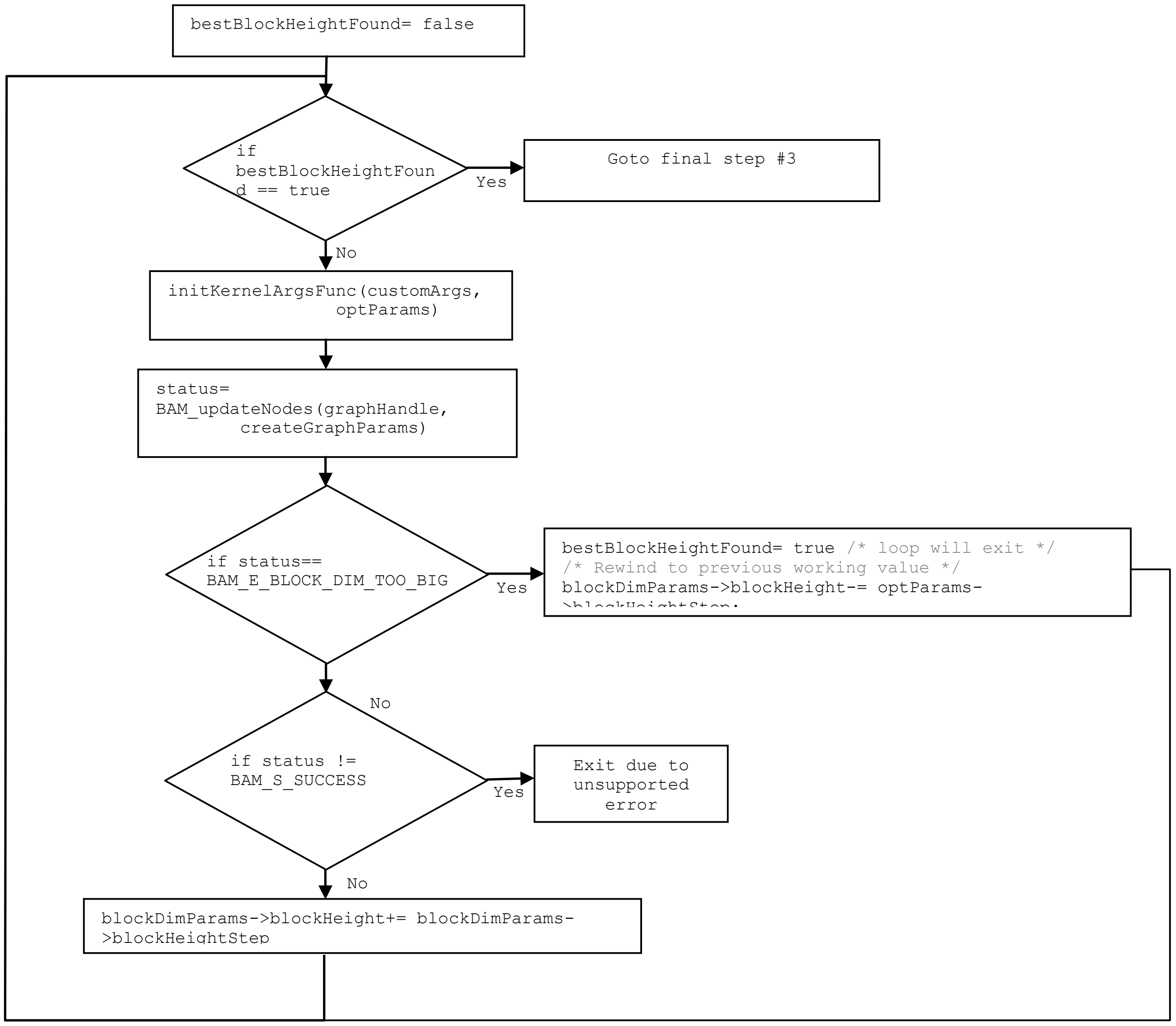

bestBlockHeightFound= false
if bestBlockHeightFound == true
Yes
Goto final step #3
No
initKernelArgsFunc(customArgs, optParams)
status= BAM_updateNodes(graphHandle, createGraphParams)
if status== BAM_E_BLOCK_DIM_TOO_BIG
Yes
bestBlockHeightFound= true /* loop will exit */
/* Rewind to previous working value */
blockDimParams->blockHeight-= optParams->blockHeightStep;
No
if status != BAM_S_SUCCESS
Yes
Exit due to unsupported error
No
blockDimParams->blockHeight+= blockDimParams->blockHeightStep

Figure 7 Flow diagram for step #2 of of the block's dimensions optimization process

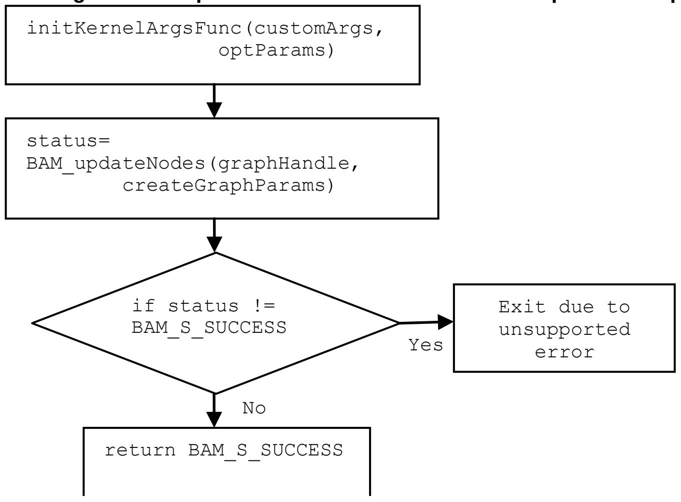

initKernelArgsFunc(customArgs, optParams)
status= BAM_updateNodes(graphHandle, createGraphParams)
if status != BAM_S_SUCCESS
Yes
Exit due to unsupported error
No
return BAM_S_SUCCESS

Figure 8 Flow diagram for step #3 of of the block's dimensions optimization process

This function optimizes the graph by maximizing the processing block width and block height while keeping all the memory record allocations succesful.

Step #1 and step #2 are almost identical except step #1 seeks to optimize the processing block width and step #2, the block height. The reason we maximize the block width first is to obtain a rectangular shape such that the EDMA bandwidth is maximized.

The search for the best block width is done by iteratively setting a bigger and bigger processing block width until the memory allocation fails. The iteration, at which the failure happens, marks the stopping point of the search and the value of the block width used in the previous iteration is the optimum one.

To populate each kernel's argument structure, the function relies on the callback function `BAM_CreateGraphParams::initKernelsArgsFunc()`:

```
typedef BAM_Status (*BAM_InitKernelsArgsFunc)(void *customArg,
BAM_BlockDimParams *blockDimParams);
```

This function, implemented by the application, is called by `BAM_createGraph()` at every iteration of the search to re-initialize arguments of the kernel that are dependent on the processing block width and height and other processing related arguments. Re-initialization of the kernel arguments at each iteration of the optimization process is required because some of these arguments are used by `BAM_KernelHelperFuncDef::getMemRec()`, which are called by `BAM_updateNodes()` within each iteration.

The prototype function `BAM_InitKernelsArgsFunc` has two input arguments:

- `BAM_BlockDimParams`

It contains the processing block dimensions and constrains that must be respected:

```
typedef struct {
    int32_t blockWidth;         /**< Processing block width */
    int32_t blockHeight;        /**< Processing block height */
    int32_t blockWidthStep;     /**< Step size used to increment the
processing block width during the optimization process */
    int32_t blockHeightStep;    /**< Step size used to increment the
processing block height during the optimization process */
    int32_t blockWidthDivisorOf; /**< Integer the blockWidth must be
divisor of, usually a processing ROI's width. If 0 then ignored */
    int32_t blockHeightDivisorOf; /**< Integer the blockHeight must be
divisor of, usually a processing ROI's height.If 0 then ignored */
    int32_t blockWidthMax; /**< @brief Integer the blockWidth must be less
than or equal to this value, usually a processing ROI's width */
    int32_t blockHeightMax; /**< @brief Integer the blockHeight must be
less than or equal to this value, usually a processing ROI's height */
} BAM_BlockDimParams;
```

Who sets the values of these members ? The application creating the graph first initializes them by setting `BAM_CreateGraphParams::blockDimParams` first. Indeed `BAM_CreateGraphParams::blockDimParams.blockWidth and BAM_CreateGraphParams::blockDimParams.blockHeight` are initialized with known working block dimensions. They can be arbitrarily low, for instance a 16x16 block dimensions will usually work as the memory requirement will be low and will easily pass at least the first iteration of the optimization process. If these initial values are too large then `BAM_createGraph()` will fail and return `BAM_E_BLOCK_DIM_TOO_BIG`.

Then, as shown in Figure 6 and Figure 7, at every iteration, `BAM_createGraph()` updates `blockDimParams->blockWidth` and `blockDimParams->blockHeight` by step of `blockDimParams->blockWidthStep` and `blockDimParams->blockHeightStep` respectively.

The rest of the members `blockWidthStep, blockHeightStep, blockWidthDivisorOf, blockHeightDivisorOf, blockWidthMax, blockHeightMax` are left untouched by `BAM` but are used to modify `blockWidth` and `blockWidth` in order to meet some SIMD alignment contraints (`blockWidthStep` and `blockHeihgtStep` are used for that) as well as some geometric constraints such as the ROI's width/height must be multiple of the block's width/height (`blockWidthDivisorOf, blockHeightDivisorOf` are used for that) while not exceeding `blockWidthMax` and `blockHeightmax`.

EVE case: `blockDimParams->blockWidthStep` **is usually set to 16 in order to align with VCOP's number of SIMD ways.** `blockDimParams->blockHeightStep` **is set to 1 or 2 to allow finer tuning of the height.**

The implementer of `BAM_InitKernelsArgsFunc` decides how to use `blockDimParams->blockWidth` and `blockDimParams->blockHeight` to set each kernel's arguments. Usually some massaging is required such as converting these dimensions into number of bytes before setting the sink and source node's arguments. Some arguments, such as threshold values, filter coefficients are not related to the block dimensions and their values take their cues from other variables that are conveyed through the other input structure `customArgs`.

- `customArgs`

The other parameter `customArg` is used to communicate information not related to the block dimensions . Within `initKernelsArgsFunc, customArg` is typecasted to a structure that it can understand.

Once the kernel arguments are initialized by `BAM_CreateGraphParams::initKernelsArgsFunc`, `BAM_createGraph()` calls the internal function `BAM_updateNodes()`, which basically update the memory partition by calling each node's `BAM_KernelHelperFuncDef::getMemRec()` helper function and then by performing memory allocation using `vcop_malloc()`.

If `BAM_updateNodes()` returns `BAM_E_BLOCK_DIM_TOO_BIG`, then we have reached the stopping point and the necessary flag is set to exit the loop. `blockDimParams->blockWidth` is also rewinded back by `blockDimParams->blockWidthStep` which is the last successful setting and becomes the optimum value.

If `BAM_updateNodes()` returns `BAM_S_SUCCESS`, then we increase `blockDimParams->blockWidth` by `blockDimParams->blockWidthStep` and we do one more iteration until `BAM_updateNodes()` returns `BAM_E_BLOCK_DIM_TOO_BIG`.

In step #2, the search for the optimum block height is performed and it works in an indentical way as the search for the optimum block width.

After step #2 is complete, both `blockDimParams->blockWidth` and `blockDimParams->blockHeight` contain some optimal values and step #3 simply calls `initKernelsArgsFunc()` and `BAM_updateNodes()` for the last time to configure the graph according to these optimum settings.

After step #3 completes, the return value of type `BAM_Status` should be `BAM_S_SUCCESS` and the memory records pointed by `createGraphParams->memRec` contains a partitioning of the memory based on the optimum block's width and height. This list of memory records can be saved to implement static allocation for the productization phase. If `BAM_E_INVALID_MEM_SIZE`is returned, that means allocated space in on-chip or external memories is too small. Remedy is to reserve larger memories and increase `createGraphParams->onChipScratchMemSize` or `createGraphParams->extMemSize`. If `BAM_E_BLOCK_DIM_TOO_BIG` is returned, then it means the initial `blockDimParams->blockWidth` and `blockDimParams->blockHeight` were too big. The fix is to reduce these initial settings.

Once the graph has been successfully created and optimized, the handle `*graphHandle` points to a valid graph object that can be passed to `BAM_process()`for execution. Actually `*graphHandle` will be exactly equal to `createParams.graphMem` which was provided as input to `BAM_createGraph()`. Once created, the graph object can be moved from external memory to internal memory, duplicated or saved into a file.

3.3 Graph Execution

The reader can refer to several examples that illustrate graph execution. The files are located in *algorithms/test/src* .

3.3.1 BAM_process()

The function `BAM_process()` is used to execute the graph:

```
int32_t BAM_process(
      BAM_GraphHandle graphHandle,
      const BAM_InArgs *inArgs,
      BAM_OutArgs *outArgs,
      const BAM_ProcessHints *hints
);
```

`graphHandle` is the handle returned by `BAM_createGraph()` and points to a graph object. Note at creation time, it is the value in `createParams.graphMem` that specificed the location of the graph object. So it is possible, that the graph object resides in slow external memory after creation. But for execution, it is preferable for the graph object to be located in internal memory. A simple memcpy or DMA copy can be used to move the graph object into faster internal memory and the new location can be passed as a `BAM_GraphHandle` to `BAM_process()`.

The input arguments `inArgs` is defined as follow:

```
typedef struct
{
  uint32_t size;      /**< Size of this structure */
  int32_t sliceIndex; /**< Index of the slice in the frame to be processed */
} BAM_InArgs;
```

`BAM_process()` accesses `inArgs->sliceIndex` to know whether it is the first slice of the frame to be processed. `inArgs->sliceIndex==0` indicates it is the first slice, any non-zero value indicates other slices. If it is the first slice, `BAM_process()` calls each kernel's `BAM_KernelExecFundDef::kernelInitFrame()` function to initialize the kernel state variables before the start of a new frame. For frame processing, caller should always set `inArgs->sliceIndex to 0`. For slice processing, the caller should increment `inArgs->sliceIndex` at every call.

The output argument `outArgs` is currently not used by `BAM_process()`, hence its dummy definition:

```
typedef struct
{
    uint32_t size;  /**< Size of this structure */
} BAM_OutArgs;
```

The input argument `hints` is an optional argument that points to a structure of type `BAM_ProcessHints`:

```
typedef struct {
    BAM_ProcessPri priority; /**< Specify whether processing schedule should
prioritize compute nodes (default) over data transfers nodes */
} BAM_ProcessHints;
```

It basically tells BAM's scheduler which node's execution is prioritized. When `BAM_ProcessHints::priority== BAM_DATA_FIRST` then source and sink node's execution is prioritized over compute node execution. This is desired if the processing is I/O bound due to high volume of the data transferred into/out of the graph or due to slow physical memory. Usually I/O bound algorithms do not take full advantage of the full processing power offered by the processor and should be avoided.

When `BAM_ProcessHints::priority== BAM_COMPUTE_FIRST` then compute node's execution is prioritized over source and sink node's execution. This is desired if the processing is compute bound .

If `hints== NULL` then BAM's default behavior is to execute a `"BAM_COMPUTE_FIRST"` schedule, that prioritizes compute nodes.
This hint option is an input arguments of `BAM_process()` rather than `BAM_createGraph()` because the type of the algorithm (I/O bound or compute bound) depends on the location of the input and output buffers passed to `BAM_process()`.
During execution of `BAM_process()`, the scheduler will:

1) Execute each kernel's initInstance and initFrame functions
`BAM_KernelExecFundDef::kernelInitInstance() &`
`BAM_KernelExecFundDef::kernelInitFrame()`. If not `NULL`, `kernelInitInstance()` function of each kernel is invoked during the first call to `BAM_process()` for a newly created graph or after a context restore. A context restore happens if the previous execution of `BAM_process()` was for a different graph. If not `NULL`, the `kernelInitFrame()` function is called if it is the first slice of the frame being processed, e.g. if `inArgs->sliceIndex==0`, to initialize the kernel state variables before the start of a new frame.

2) Execute each kernel's compute and wait functions
`BAM_KernelExecFundDef::kernelCompute()` and
`BAM_KernelExecFundDef::kernelWait()` as many times as necessary to consume all the blocks in the frame. For optimum performance, the execution schedule assumes source and sink nodes compute functions can be executed in parallel with the compute nodes' function. The wait functions only apply for the sink and source nodes.

3.4 Controlling graph's nodes after graph creation

From BAM 2.3.0 a new function called `BAM_controlNode()` allows to dynamically change a graph node's execution parameters after the graph is created and before each process call, as long as a the control function for the associated kernel is provided.

The API of `BAM_controlNode()` is as follow:

```
BAM_Status BAM_controlNode(
      BAM_GraphHandle graphHandle,
      uint8_t nodeIndex,
      void * controlArgs
);
```

`BAM_controlNode()` can either query or modify the value of one or several arguments associated to the node referred by `nodeIndex` composing the graph pointed by `graphHandle`.
`nodeIndex` would correspond to one of the identifier used when specifying the node list `BAM_CreateGraphParams::nodeList` during graph creation.
Actually `BAM_controlNode()` just calls the `BAM_KernelExecFundDef::kernelCtl()` of the kernel associated to the node. In some way, it acts as a wrapper, by looking up the function to call and passing the generic argument structure `controlArgs`. The underlying behavior, e.g. whether to query of modify any argument values is implemented by the kernel execute function `BAM_KernelExecFundDef::kernelCtl()` which has the following prototype:

```
typedef uint32_t (*BAM_KernelCtlFunc)(void *kernelContext, void *
controlArgs);
```

`BAM_controNode()` calls this function. The argument `kernelContext` is determined internally by `BAM_controlNode()` and `controlArgs` is the same pointer that is passed to `BAM_controlNode()`.
It points to a custom control argument structure, specific to a kernel. For example, the non maximum suppression for signed 32-bits kernel has the following control argument structure defined in `kernels/vlib/vcop_nonmaxsuppress_mxn_s32/bam_helper`:

```
typedef struct
{
    uint8_t ctlCmdId;
    int32_t threshold;
} BAM_NonMaxSuppress_mxn_S32_CtlArgs;
```

The `ctlCmdId` can be set to one of the following value:

```
#define BAM_NONMAXSUPPRESS_MxN_S32_CMD_GET_THRESHOLD 0
#define BAM_NONMAXSUPPRESS_MxN_S32_CMD_SET_THRESHOLD 1
```

We can infer that `BAM_KernelExecFundDef::kernelCtl()` implementation will be able to set the NMS threshold to a new value passed in `threshold`, if `ctlCmdId== BAM_NONMAXSUPPRESS_MxN_S32_CMD_SET_THRESHOLD`. Otherwise, if `ctlCmdId== BAM_NONMAXSUPPRESS_MxN_S32_CMD_GET_THRESHOLD`, the control function will simply return the current threshold value.
The implementation is given below:

```
static uint32_t BAM_nonMaxSuppress_mxn_S32_control(void *kernelContext, void
* controlArgs){
    BAM_NonMaxSuppress_mxn_S32_Context *context =
(BAM_NonMaxSuppress_mxn_S32_Context *) kernelContext;
    BAM_NonMaxSuppress_mxn_S32_CtlArgs *ctlArg=
(BAM_NonMaxSuppress_mxn_S32_CtlArgs *) controlArgs;
    uint32_t status = 0;

    if (ctlArg->ctlCmdId== BAM_NONMAXSUPPRESS_MxN_S32_CMD_SET_THRESHOLD){
        context->kernelArgs.threshold= ctlArg->threshold;
    }
    else if (ctlArg->ctlCmdId==BAM_NONMAXSUPPRESS_MxN_S32_CMD_GET_THRESHOLD){
        ctlArg->threshold= context->kernelArgs.threshold;
    }
    else {
        status= 1;
    }

    return status;
}
```

We see typecasting of `ctlArg` is necessary since each kernel can have its own control argument structure.

EVE case: The implementer of any `BAM_KernelExecFundDef::kernelCtl()` **should respect the following rule: the function can modify any members inside the context structure using values passed in the** `controlArgs` **structure. However, it cannot touch any location in VCOP mem such as WMEM, IMGBUF. Modification of the VCOP mem is eventually done by the** `BAM_KernelInitFrame()` **function, called by** `BAM_process()` **at every frame boundary.**

3.5 Development/Debugging tips

3.5.1 Implement initial proprotype in pure C code

Since a graph can be composed of nodes entirely implemented in C, a golden C version of each kernel should be first implemented and the graph be implemented based on these. Once the C-only graph is fully debugged, one can develop the kernel-C version of each node and simply swap the nodes one after the other until the full graph runs on VCOP. Note that the kernel database will initially have both versions of the kernels but for production, only the kernel-C implementions need to be there.

With BAM software release, the ISP example code has the following kernels that can be swapped with the natural C version: white balance gains and gamma correction. To enable the natural-C version, simply update the `NODELIST` array in *`modules/algorithms/src/isp/src/isp_graph.c:`*

```
static const BAM_NodeParams NODELIST[]= {\
  {SOURCE_NODE, BAM_TI_KERNELID_DMAREAD_AUTOINCREMENT, NULL},\
  {SUBSAMPLING_NODE, BAM_TI_KERNELID_SUBSAMPLING, NULL},\
  {WHITEBALANCE_GAINS_NODE, BAM_TI_KERNELID_APPLY_WHITEBALANCE_GAINS, NULL},
  {BAYER2RGB_NODE, BAM_TI_KERNELID_BAYER2RGB, NULL},\
  {RGB2SRGB_NODE, BAM_TI_KERNELID_RGB2SRGB, NULL},\
  {GAMMACORRECTION_NODE, BAM_TI_KERNELID_GAMMACORRECTION, NULL},
  {RGB2YUV_NODE, BAM_TI_KERNELID_RGB2YUV, NULL},\
  {SINK_NODE,BAM_TI_KERNELID_DMAWRITE_AUTOINCREMENT, NULL},\
  {BAM_END_NODE_MARKER, 0, NULL},\
};
```

To enable the kernel-C version, simply replace `BAM_TI_KERNELID_APPLY_WHITEBALANCE_GAINS` with `BAM_TI_KERNELID_APPLY_NATCWHITEBALANCE_GAINS` and `BAM_TI_KERNELID_GAMMACORRECTION` with `BAM_TI_KERNELID_NATCGAMMACORRECTION`. The database definition in file *`modules/kernels/bamdb/bam_kernel_database`* contains reference to both the kernel-C(VCOP) and C (ARP32) versions of these kernels.

3.5.2 *Debugging the graph creation process*

The memory partitioning can be checked after the execution of

```
BAM_createGraph(
  BAM_CreateGraphParams *graphCreateParams,
  BAM_GraphHandle *pGraphHandle);
```

by accessing the member `graphCreateParams->memRec` .

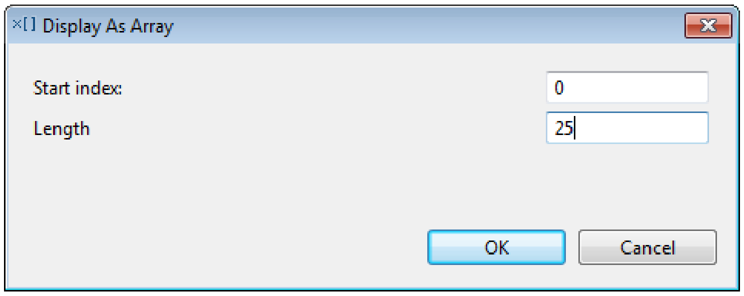
Follow these steps:

1) Put a breakpoint after `BAM_createGraph()` .

2) Run up to the breakpoint.

3) Open the expression window and add `graphCreateParams.numMemRec` to it. Note its value.

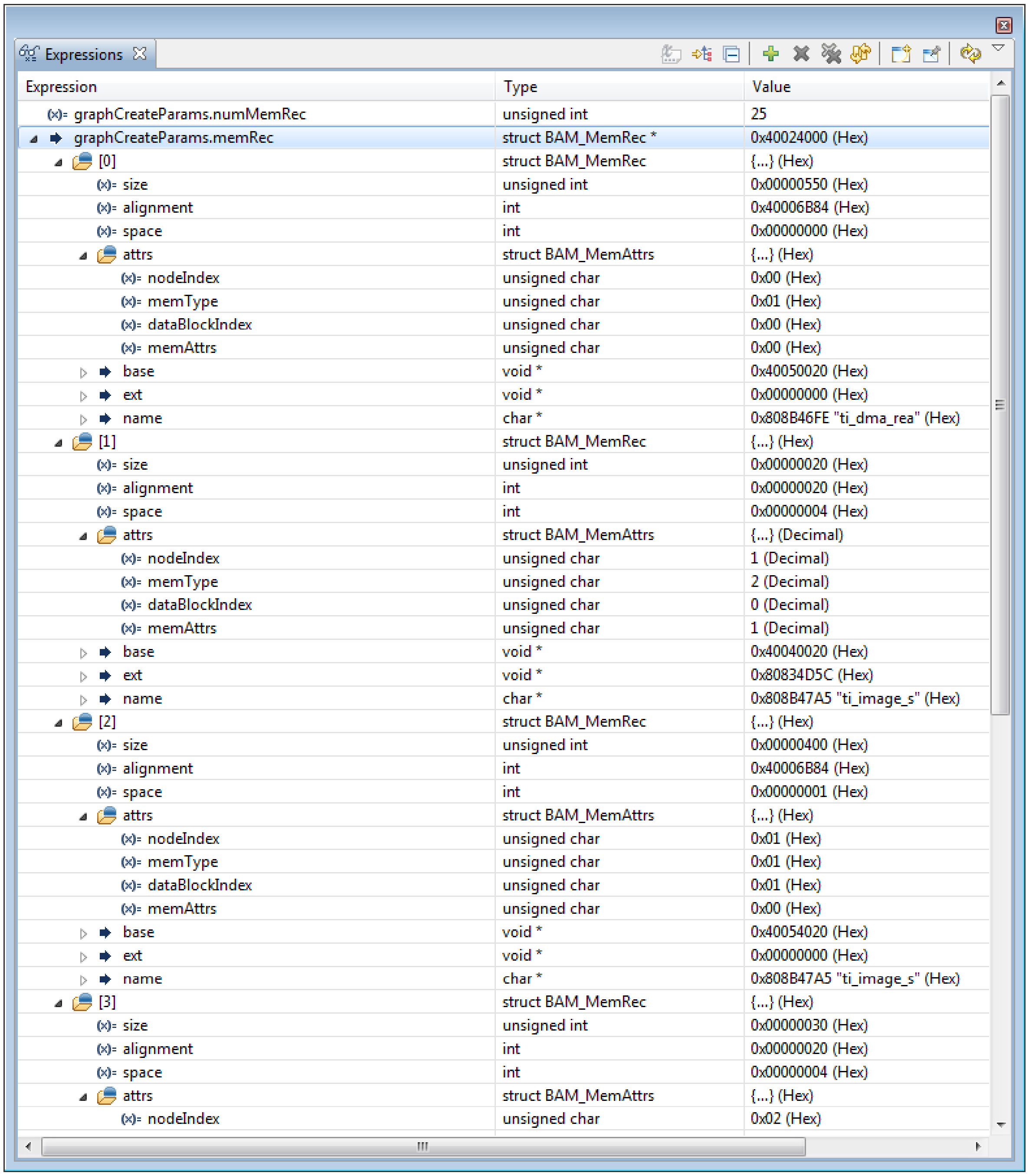
4) Add the variable `graphCreateParams.memRec` to the expression window and right click on it to select ‘Display as Array’. Set ‘Length’ to the value set in `graphCreateParams.numMemRec` .



Start index: 0
Length 25

5) Right-click on `graphCreateParams.memRec`, select Number Format and then Hex.

6) Inspect each memory record by expanding it. Use the member `name` as a hint to what node the memory record corresponds to. The member `attrs.memType` indicates whether the memory is for input (0), output (1) or internal (2) usage and `attrs.memAttrs` tells whether it is allocated as scratch (0), persistent (1) or constant (2) memory.



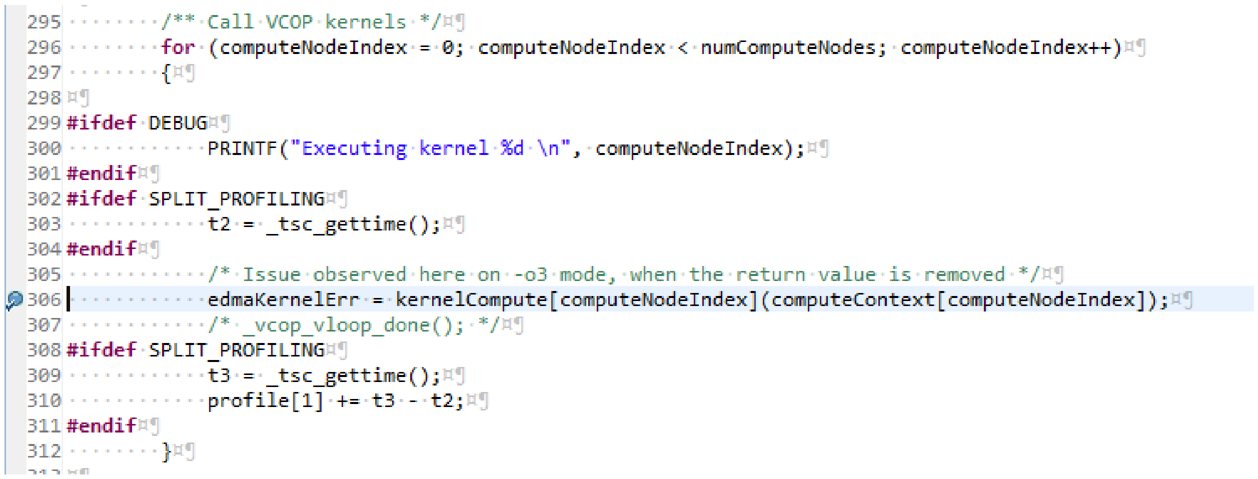
Expression	Type	Value
graphCreateParams.numMemRec	unsigned int	25
graphCreateParams.memRec	struct BAM_MemRec *	0x40024000 (Hex)
[0]	struct BAM_MemRec	{...} (Hex)
size	unsigned int	0x00000550 (Hex)
alignment	int	0x40006B84 (Hex)
space	int	0x00000000 (Hex)
attrs	struct BAM_MemAttrs	{...} (Hex)
nodeIndex	unsigned char	0x00 (Hex)
memType	unsigned char	0x01 (Hex)
dataBlockIndex	unsigned char	0x00 (Hex)
memAttrs	unsigned char	0x00 (Hex)
base	void *	0x40050020 (Hex)
ext	void *	0x00000000 (Hex)
name	char *	0x808B46FE "ti_dma_rea" (Hex)
[1]	struct BAM_MemRec	{...} (Hex)
size	unsigned int	0x00000020 (Hex)
alignment	int	0x00000020 (Hex)
space	int	0x00000004 (Hex)
attrs	struct BAM_MemAttrs	{...} (Decimal)
nodeIndex	unsigned char	1 (Decimal)
memType	unsigned char	2 (Decimal)
dataBlockIndex	unsigned char	0 (Decimal)
memAttrs	unsigned char	1 (Decimal)
base	void *	0x40040020 (Hex)
ext	void *	0x80834D5C (Hex)
name	char *	0x808B47A5 "ti_image_s" (Hex)
[2]	struct BAM_MemRec	{...} (Hex)
size	unsigned int	0x00000400 (Hex)
alignment	int	0x40006B84 (Hex)
space	int	0x00000001 (Hex)
attrs	struct BAM_MemAttrs	{...} (Hex)
nodeIndex	unsigned char	0x01 (Hex)
memType	unsigned char	0x01 (Hex)
dataBlockIndex	unsigned char	0x01 (Hex)
memAttrs	unsigned char	0x00 (Hex)
base	void *	0x40054020 (Hex)
ext	void *	0x00000000 (Hex)
name	char *	0x808B47A5 "ti_image_s" (Hex)
[3]	struct BAM_MemRec	{...} (Hex)
size	unsigned int	0x00000030 (Hex)
alignment	int	0x00000020 (Hex)
space	int	0x00000004 (Hex)
attrs	struct BAM_MemAttrs	{...} (Hex)
nodeIndex	unsigned char	0x02 (Hex)

3.5.3 *Debugging the graph execution process*

The graph processing is kicked off inside `BAM_process()` call and after the function returns, the processing will have completed for an entire frame/slice. Under the hood `BAM_process()` calls `BAM_execute()`, which handles the scheduling between the DMA-based source and sink nodes with the compute nodes. It is actually possible to put a breakpoint at a specific location in `BAM_execute()` and then have the debugger halt at every kernel computation function.

1) Open the file *algframework/src/bam/bam_execute.c*

2) Locate line 306 and put a breakpoint there:



```
295         /** Call VCOP kernels */
296         for (computeNodeIndex = 0; computeNodeIndex < numComputeNodes; computeNodeIndex++)
297         {
298 
299 #ifdef DEBUG
300             PRINTF("Executing kernel %d \n", computeNodeIndex);
301 #endif
302 #ifdef SPLIT_PROFILING
303             t2 = _tsc_gettime();
304 #endif
305             /* Issue observed here on -o3 mode, when the return value is removed */
306             edmaKernelErr = kernelCompute[computeNodeIndex](computeContext[computeNodeIndex]);
307             /* _vcop_vloop_done(); */
308 #ifdef SPLIT_PROFILING
309             t3 = _tsc_gettime();
310             profile[1] += t3 - t2;
311 #endif
312         }
```

3) Run the executable and once it halts at the breakpoint, step into the `kernelCompute[computeNodeIndex](computeContext[computeNodeIndex])` function. On EVE, if a particular node's `kernelCompute` member of structure `BAM_KernelExecFuncDef` was provided as NULL then the code will step into the `vcop_<kernel_name>_vloops()` directly, which was used to initialize the `kernelCustom` member. Please refer to paragraph 3.2.1.3 for more details. On the other hand, if `kernelCompute` was set to point to a valid implementation, then the code will step into it.

4) It is possible to view the values of the members of a kernel context structure by adding `computeContext[computeNodeIndex]` to the expression window and type casting it to the associated custom structure type. But in the EVE case, this will work only if `kernelCompute` of `BAM_KernelExecFuncDef` was set to non-NULL. If set to `NULL`, `computeContext[computeNodeIndex]` contains the pointer to the param registers block instead of the context structure of the node.

5) To view the content of the VCOP memory, (image buffers, wbuf), make sure that access is granted to ARP32 by executing the gel function *ToggleBuffers* in sub-menu VCOP Control of the Code Composer Script menu. You need to have the latest *eden_arp32.gel* loaded in order to access this function. After viewing the content of these memories, don't forget to toggle these buffers back to the original access configuration by executing *ToggleBuffers* again.
You should see this message in the console window when toggling the access to the system (ARP32):

```
ARP32_0: GEL Output: VCOP buffers are unlocked for system view.
```

and this message when toggling back to VCOP:

```
ARP32_0: GEL Output: VCOP buffers are locked to application control.
```

3.5.4 Typical issues encountered

If the stack is not large enough, some unexpected behavior can happen. BAM v2.1 needs at least 0x1200h bytes of stack so make sure that your linker command file or CCS project reserves enough space.

3.6 Profiling

BAM automatically collects execution time of the following stages that happen within the creation and the execution of a graph:

a) Function `BAM_createGraph()` which includes the execution of: the smart memory allocator, the optimum block dimension optimizer, the callback function BAM_CreateParams::initKernelsArgsFunc, all kernels' `BAM_KernelHelperFuncDef::getMemRec()` and `BAM_KernelHelperFuncDef::setMemRec()` functions.

b) The following function called by `BAM_process()`: `BAM_KernelExecFundDef::kernelCtl()`, `BAM_KernelExecFuncDef::kernelInitInstance`, `BAM_KernelExecFuncDef::kernelInitFrame`. The resulting timing information is equal to the execution time of all these functions clubbed together. There is no granular measurement done in order to keep the overall execution overhead low since these functions are called within `BAM_process()`.

c) The following function called by `BAM_process()`: `BAM_KernelExecFundDef::kernelCompute()` and `BAM_KernelExecFundDef::kernelWait`(). Same observation as above, the measurements are made for the whole group of functions, not individually, in order to keep the timing infrastructure overhead low inside `BAM_process()`.

On one hand, since `BAM_createParams()` is called one time, only one occurrence of timing measurement for a) is obtained by BAM. On the other hand, `BAM_process()` is called many times in real-world use case, thus many measurements for b) and c) are done. One probably would like to know how many times, these functions were executed and what is their average, maximum, minimum runtime. The function `BAM_getStats()` provides these statistics.

```
BAM_Status BAM_getStats(BAM_GraphHandle graphHandle, BAM_Statistics *stat);
```

`graphHandle` identify for which graph, these statistics need to be returned for.
`stat` points to the output statistic structure:

```
typedef struct {

    BAM_TimingInfo createGraph; /**< Stats corresponding to BAM_createGraph()
*/

    BAM_TimingInfo kernelsInit; /**< Stats corresponding to the execution of
all kernelCtl(), kernelInitInstance(), kernelInitFrame() functions in the
graph */

    BAM_TimingInfo kernelsExec; /**< Stats corresponding to the execution of
all kernelCompute(), kernelWait() functions in the graph */

} BAM_Statistics;
```

with type `BAM_TimingInfo`:

```
typedef struct {

    uint32_t numExecs; /**< Number of times this stage was executed,
CURRENTLY DISABLED */

    uint32_t minCycles; /**< minimum of all the execution times collected so
far, CURRENTLY DISABLED  */
```

```
    uint32_t maxCycles; /**< minimum of all the execution times collected so
far, CURRENTLY DISABLED   */

    uint32_t averCycles; /**< average of all the execution times collected so
far, CURRENTLY DISABLED   */

    uint32_t lastCycles; /**< execution time of the last execution */

} BAM_TimingInfo;
```

Note that in current implementation, only `lastCycles` is filled out by `BAM_getStats()`.

3.7 Library Version

The version number of the BAM library linked with the application can be obtained at runtime by executing the function `BAM_getVersion()`:

```
BAM_Status BAM_getVersion(uint16_t *major, uint16_t *minor, uint16_t *patch);
```

This function returns the major, minor and patch version numbers into the corresponding placeholders passed as arguments. These version numbers are macro defined in the file *bam_common.h .*

Chapter 4

API Reference

The API references are available in form of html files auto-generated by doxygen.

The reader should access the documentation at *algframework/doc/bam_algframework.chm*

Chapter 5

Current limitations

5.1 Source and Sink nodes

Current implementation of source and sink nodes rely on EDMA for quick transfers. One example DMA node to perfom 2D auto increment transfer is supplied as part of Bam framework. User can create custom nodes for different transfer patterns.

Up to one source node and one sink node is allowed per block. Multiple transfers are handled by setting up multiple ports at each node.

5.2 Order kernels are listed in the database

A kernel database `BAM_KernelHostDBdef` or `BAM_KernelExecDBdef` must list all the kernels in the order specified by their respective kernelID, which must follow a sequentially increasing order. The function `BAM_initKernelDB()` will detect any violation to this condition and returns the error code `BAM_E_DATABASE_KERNELID`.

5.3 Specifications of edge list

If one single port of an upstream node has more than one connection, which happens in case of a fork in the graph, then all these connections must be clustered together. For instance in the below image pyramid graph, every `DS_NODE'`s output port `BAM_BLOCKAVERAGE2x2_OUTPUT_PORT` has two connections: one to the next `DS_NODE` and one to `SINK_NODE.` These connections must appear one afte the other in edge list. If there is a connection involving another node or another port between them then graph creation would be incorrect.

This is a valid edge list:

```
BAM_EdgeParams EDGELIST[]= {
{{SOURCE_NODE,  BAM_SOURCE_NODE_PORT1},            {DS_NODE1,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},
{{DS_NODE1,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {SINK_NODE, BAM_SINK_NODE_PORT1}},\
{{DS_NODE1,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {DS_NODE2,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE2,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {SINK_NODE, BAM_SINK_NODE_PORT2}},\
{{DS_NODE2,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {DS_NODE3,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE3,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {SINK_NODE, BAM_SINK_NODE_PORT3}},\
{{DS_NODE3,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {DS_NODE4,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE4,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {SINK_NODE, BAM_SINK_NODE_PORT4}},\
{{DS_NODE4,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {DS_NODE5,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE5,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},   {SINK_NODE, BAM_SINK_NODE_PORT5}},\
{{BAM_END_NODE_MARKER, 0},                          {BAM_END_NODE_MARKER, 0}},\
};
```

This is an invalid edge list because for every `DS_NODE`, each two connections originating from `BAM_BLOCKAVERAGE2x2_OUTPUT_PORT` are separated:

```
BAM_EdgeParams EDGELIST[]= {
{{SOURCE_NODE,  BAM_SOURCE_NODE_PORT1},           {DS_NODE1,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},
{{DS_NODE1,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {SINK_NODE, BAM_SINK_NODE_PORT1}},\
{{DS_NODE2,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {SINK_NODE, BAM_SINK_NODE_PORT2}},\
{{DS_NODE3,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {SINK_NODE, BAM_SINK_NODE_PORT3}},\
{{DS_NODE4,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {SINK_NODE, BAM_SINK_NODE_PORT4}},\
{{DS_NODE1,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {DS_NODE2,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE2,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {DS_NODE3,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE3,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {DS_NODE4,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE4,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {DS_NODE5,  BAM_BLOCKAVERAGE2x2_INPUT_PORT}},\
{{DS_NODE5,     BAM_BLOCKAVERAGE2x2_OUTPUT_PORT},  {SINK_NODE, BAM_SINK_NODE_PORT5}},\
{{BAM_END_NODE_MARKER, 0},                        {BAM_END_NODE_MARKER, 0}},\
};
```

5.4 BAM_process()

Currently only one schedule is supported: the default one which is a `"BAM_COMPUTE_FIRST"` schedule, that prioritizes compute nodes. Consequently setting `hints->priority` to `BAM_DATA_FIRST` does not change the processing schedule.

When compute nodes fully implemented on ARP32 and accessing the image buffers are present in the graph, these nodes do not run concurrently with other nodes implemented on VCOP or EDMA. Such nodes are really adequate for post-processing or during development stage where a full natural C version of a graph needs to be developed.
This restriction does not apply to source and sink nodes that use EDMA as hardware resources, these nodes operate in parallel with the compute node that are implemented on VCOP.

5.5 BAM_getStats()

Currently the function will only return the statistics of the last graph created/executed. In fact, it does not take into account the value of the parameter `graphHandle`. Also only the member `lastCycles` in `BAM_TimingInfo` is filled out.

Chapter 6

Frequently Asked Questions

1) Can you quantify the performance difference with and without BAM ?

We used resizer applet to illustrate the performance differences under 4 different scenarios.

The non-BAM version always process blocks of 64x32 pixels whereas the BAM version can find more optimum block dimensions depending on the flag optimizeBlockDim.

- Scenario #1: optimizeBlockDim= false, processing blocks of 64x32

Stages	BAM Applet MegaCyc	Barebones Applet MegaCyc	Comments
Graph creation / memory allocation	0.07	0.007	BAM uses smart memory allocation whereas barebones applet uses vcop_malloc()
Kernel and EDMA init	0.02	0.01	BAM DMA nodes have more initialization overhead than EDMA starterware functions
Graph/Applet execution	0.88	1.05	BAM_process() function is very well optimized
Total	0.97	1.07	Despite graph creation overhead, still in overall BAM applet manages to perform better than barebone implementation.

- Scenario #2: optimizeBlockDim= true, 64x32 is initial hint for the block dimensions optimizer. Final block size is 240x90

Stages	BAM Applet MegaCyc	Barebones Applet MegaCyc	Comments
Graph creation / memory allocation	1.06	0.007	The block dimensions optimizer consumes a lot of cycles but that's because the initial hint is too far.

Kernel and EDMA init	0.02	0.01	BAM DMA nodes have more initialization overhead than EDMA starterware functions
Graph/Applet execution	0.64	1.05	BAM could figure out larger block dimensions. Barebone still uses 64x32.
Total (excluded graph creation which can be done statically)	0.66	1.06	

- Scenario #3: optimizeBlockDim= true, 240x44 is initial hint for the block dimensions optimizer. Final block size is 240x90

Stages	BAM Applet MegaCyc	Barebones Applet MegaCyc	Comments
Graph creation / memory allocation	0.7	0.007	Having a better initial hint helps in reducing graph creation time
Kernel and EDMA init	0.02	0.01	BAM DMA nodes have more initialization overhead than EDMA starterware functions
Graph/Applet execution	0.64	1.05	BAM could figure out larger block dimensions. Barebone still uses 64x32.
Total (excluded graph creation which can be done statically or at startup)	0.66	1.06	

- Scenario #4: optimizeBlockDim= true, 240x90 is initial hint for the block dimensions optimizer. Final block size is 240x90

Stages	BAM Applet MegaCyc	Barebones Applet MegaCyc	Comments
Graph creation / memory allocation	0.14	0.007	We use an initial hint equal to the best block dimension

Kernel and EDMA init	0.02	0.01	BAM DMA nodes have more initialization overhead than EDMA starterware functions
Graph/Applet execution	0.64	1.05	BAM uses 240x90 blocks whereas Barebone still uses 64x32.
Total (excluded graph creation which can be done statically or at startup)	0.66	1.06	

In conclusion, although graph creation takes additional cycles compared to barebones implementation, it is only a one-time hit incurred at startup time. Besides a graph can be created during algorithm prototyping and production code can just use the statically linked object. In this case, no graph creation time is incurred.

2) What is the development effort with and without BAM ?
Kernel development time will be little higher, because the developer has to write the helper functions and verify it (additional 0.5 day). Integration/Applet development time will be much faster - especially because you don’t really have to worry about buffer allocations and the flow of buffers from one kernel to other.

3) How is the Testing/Debugging effort with and without BAM ?

BAM user’s guide has a section on debugging tips, which shows how easy it is to step through the execution kernel of each node.

BAM has scope of improving its debug capability by adding debug, trace and profile messages.

The way it should be seen is - if there are multiple developers and they want to exchange their code and might need to debug other’s code – debug effort is easier with BAM because the code flow and data structures are standardized and common across all the implementations - so familiarity will be higher compared to individual’s way of implementing.
If there are 1-2 developers and they have already defined routines – may be there will be inertia to learn a different way.

4) What is the ramp up effort for a new developer ?
Kernel developer needs 3 days to ramp up.
Applet developer needs 2 days to ramp up.

Chapter 7

Example code

The following is some example code for the `initKernelArgsFunc()` callback function, which is a member of the structure `BAM_CreateGraphParams` passed to `BAM_createGraph()`.

This particular example concerns the arrayOp graph example and was extracted from the file *`algorithms\src\array_op\src\array_op_graph.c`* .

```
int32_t arrayOp_initKernelsArgsFunc(void *initKernelsArgs, BAM_BlockDimParams
*blockDimParams) {
    int16_t blkWidthFinal= 0;
    int16_t blkHeightFinal= 0;

    int32_t status = ARRAY_OP_SUCCESS;

    ArrayOpInitKernelsArgs *arrayOpInitKernelsArgs=
(ArrayOpInitKernelsArgs*)initKernelsArgs;
    arrayOp_graphObj *graphObj= arrayOpInitKernelsArgs->graphObj;
    arrayOp_CreateParams *createParams= arrayOpInitKernelsArgs->createParams;

    EDMA_UTILS_autoIncrement_initParam *dmaReadKernelArgs = &graphObj-
>dmaReadKernelArgs;
    EDMA_UTILS_autoIncrement_initParam *dmaWriteKernelArgs = &graphObj-
>dmaWriteKernelArgs;
    BAM_Image_arrayOp_Args *arrayOpkernelArgs = &graphObj->arrayOpkernelArgs;

    blkWidthFinal = blockDimParams->blockWidth;
    blkHeightFinal = blockDimParams->blockHeight;

    if ((createParams->imgSliceWidth % blkWidthFinal) != 0)
    {
        PRINTF("imgSliceWidth = %d, must be a multiple of %d!!!",createParams-
>imgSliceWidth, blkWidthFinal);
        status = ARRAY_OP_FAIL;
        goto Exit;
    }
    if ((createParams->imgSliceHeight % blkHeightFinal) != 0)
    {
        PRINTF("imgSliceHeight = %d, must be a multiple of %d!!!",createParams-
>imgSliceHeight, ARRAY_OP_BLK_HEIGHT);
        status = ARRAY_OP_FAIL;
        goto Exit;
    }
    dmaReadKernelArgs->numInTransfers                                         = 2;
    dmaReadKernelArgs->transferType                                           =
EDMA_UTILS_TRANSFER_IN;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].imgFrameWidth      =
createParams->imgFrameWidth;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].imgFrameHeight     =
createParams->imgFrameHeight;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiWidth           =
createParams->imgFrameWidth;
```

```
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiHeight            =
createParams->imgFrameHeight;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkWidth             =
blkWidthFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkHeight            =
blkHeightFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkIncrementX        =
blkWidthFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkIncrementY        =
blkHeightFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiOffset            = 0;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].extMemPtr            =
NULL;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].extMemPtrStride      =
createParams->imgSliceWidth;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].interMemPtr          =
NULL;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].interMemPtrStride    =
blkWidthFinal;

    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].imgFrameWidth        =
createParams->imgFrameWidth;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].imgFrameHeight       =
createParams->imgFrameHeight;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].roiWidth             =
createParams->imgFrameWidth;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].roiHeight            =
createParams->imgFrameHeight;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].blkWidth             =
blkWidthFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].blkHeight            =
blkHeightFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].blkIncrementX        =
blkWidthFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].blkIncrementY        =
blkHeightFinal;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].roiOffset            = 0;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].extMemPtr            =
NULL;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].extMemPtrStride      =
createParams->imgSliceWidth;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].interMemPtr          =
NULL;
    dmaReadKernelArgs->transferProp[BAM_SOURCE_NODE_PORT2].interMemPtrStride    =
blkWidthFinal;

    dmaWriteKernelArgs->numOutTransfers                                          = 1;
    dmaWriteKernelArgs->transferType                                             =
EDMA_UTILS_TRANSFER_OUT;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].imgFrameWidth        =
createParams->imgFrameWidth*sizeof(short);
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].imgFrameHeight       =
createParams->imgFrameHeight;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiWidth             =
createParams->imgSliceWidth*sizeof(short);
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiHeight            =
createParams->imgFrameHeight;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkWidth             =
blkWidthFinal*sizeof(short);
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkHeight            =
blkHeightFinal;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkIncrementX        =
blkWidthFinal*sizeof(short);
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].blkIncrementY        =
blkHeightFinal;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].roiOffset            = 0;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].extMemPtr            =
NULL;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].extMemPtrStride      =
createParams->imgSliceWidth*sizeof(short);
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].interMemPtr          =
NULL;
    dmaWriteKernelArgs->transferProp[BAM_SOURCE_NODE_PORT1].interMemPtrStride    =
blkWidthFinal*sizeof(short);
    /*------------------------------------------------------------------------*/
    /*Params update for array operation
*/
    /*------------------------------------------------------------------------*/
    arrayOpkernelArgs->inputA_blk_width     = blkWidthFinal;
    arrayOpkernelArgs->inputA_blk_height    = blkHeightFinal;
    arrayOpkernelArgs->inputB_blk_width     = blkWidthFinal;
    arrayOpkernelArgs->inputB_blk_height    = blkHeightFinal;
    arrayOpkernelArgs->output_blk_width     = blkWidthFinal;
    arrayOpkernelArgs->output_blk_height    = blkHeightFinal;
    arrayOpkernelArgs->compute_blk_width    = blkWidthFinal;
    arrayOpkernelArgs->compute_blk_height   = blkHeightFinal;
    arrayOpkernelArgs->rnd_shift            = createParams->rnd_shift;

    graphObj->nodeList[SOURCE_NODE].kernelArgs= (void *) dmaReadKernelArgs;
    graphObj->nodeList[SINK_NODE].kernelArgs= (void *) dmaWriteKernelArgs;
    graphObj->nodeList[ARRAY_OP_NODE].kernelArgs= (void *) arrayOpkernelArgs;

    Exit:
    return (status);
}
```

Chapter 8

Differences between BAM v2.02 and BAM v2.03

8.1 API changes in BAM_process()

In BAM 2.02, the function `BAM_process()` has the following prototype:

```
int32_t BAM_process(
      BAM_GraphHandle graphHandle,
      const BAM_InBufs *inBufs,
      const BAM_OutBufs *outBufs,
      const BAM_InArgs *inArgs,
      BAM_OutArgs *outArgs,
      const BAM_ProcessHints *hints
);
```

Whereas in BAM 2.03, the two input arguments `inBufs and outBufs` have been removed from the prototype, which has become:

```
int32_t BAM_process(
      BAM_GraphHandle graphHandle,
      const BAM_InArgs *inArgs,
      BAM_OutArgs *outArgs,
      const BAM_ProcessHints *hints
);
```

In BAM 2.02, the purpose of `inBufs` and `outBufs` was to hold the descriptions of the input and output frame buffers including pointers in external memory, stride, width and height. `BAM_process()` would transfer these settings into the source and sink nodes by calling these nodes' associated control functions `BAM_KernelExecFundDef::kernelCtl()`.
In BAM 2.03, since these arguments are removed, any changes to the source and sink node transfer parameters is done by calling `BAM_controlNode()` prior to `BAM_process()`. Calling `BAM_controlNode()` has become the standard way to change a node's parameters.

A direct consequence is that every applet now implements a function `<APPLET>_TI_dmaControl()` in file *<APPLET>_graph.c*, which relies on `BAM_controlNode()` to update transfer parameters of the source and sink nodes. This function can be called whenever an applet needs to update transfer parameters related to a source or sink node.
The next paragraph explains how to port an applet whas has been developed using BAM 2.02 to use BAM 2.03.

8.2 Applet porting steps to use new BAM_process()

8.2.1 Implementation of <APPLET>_TI_dmaControl()

This function is implemented in file *<APPLET>_graph.c,* is customed to the applet and calls `BAM_controlNode()` to change the transfer parameters passed to the sink and source nodes. It first translates relevant members present in *the IVISION_BufDesc* structures to control parameters expected by the source and sink nodes.

Below is a simple example for the image pyramid applet:

```
 int32_t IMAGE_PYRAMID_U8_TI_dmaControl(const BAM_GraphMem *graphMem, const
IMAGE_PYRAMID_U8_TI_Handle intAlgHandle, const IVISION_BufDesc *inBufDesc, const
IVISION_BufDesc *outBufDesc)
{
    EDMA_UTILS_autoIncrement_updateParams autoIncCtlArgs;

    const IVISION_BufPlanes * ivisionBufPlane= &inBufDesc->bufPlanes[0];
    int32_t status = BAM_S_SUCCESS;
    uint16_t extraBorders= 0;
    uint8_t i=0;

    /* initilize the sub-handles inside the main handle */
    BAM_GraphHandle graphHandle = (BAM_GraphHandle)graphMem->graphObj;

    /* Initialize the control parameters for the SOURCE auto-increment DMA node */
    autoIncCtlArgs.updateMask= EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTR |
EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTRSTRIDE;

    /* In case gaussian filter is used, we rewind topLeft.x and topLeft.y by the exact
border width and height required for a 5x5 gaussian
     * This is to ensure that the pointer for EDMA read points exactly to the upper
left corner of the region composed
     * by the ROI augmented by the filtering border.
     * */
    if (intAlgHandle->filterType== IMAGE_PYRAMID_U8_TI_5x5_GAUSSIAN) {

        extraBorders= 0;
        for (i=0;i<intAlgHandle->numLevels;i++){
            extraBorders+= (1<<i)*4;
        }

        autoIncCtlArgs.updateParams[0].extMemPtr= (uint8_t*)ivisionBufPlane->buf +
((ivisionBufPlane->frameROI.topLeft.y - extraBorders/2) * ivisionBufPlane->width +
(ivisionBufPlane->frameROI.topLeft.x - extraBorders/2));
        autoIncCtlArgs.updateParams[0].extMemPtrStride = ivisionBufPlane->width;

    }
    else {
        autoIncCtlArgs.updateParams[0].extMemPtr= (uint8_t*)ivisionBufPlane->buf +
(ivisionBufPlane->frameROI.topLeft.y * ivisionBufPlane->width) + ivisionBufPlane-
>frameROI.topLeft.x;
        autoIncCtlArgs.updateParams[0].extMemPtrStride = ivisionBufPlane->width;
    }

    status= BAM_controlNode(graphHandle, SOURCE_NODE, &autoIncCtlArgs);

    if (status!= BAM_S_SUCCESS) {
        goto Exit;
    }
    /* Initialize the control parameters for the SINK auto-increment DMA node */
    ivisionBufPlane= &outBufDesc->bufPlanes[0];

    autoIncCtlArgs.updateMask= EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTR |
EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTRSTRIDE;

    for (i=0;i<intAlgHandle->numLevels;i++){
        autoIncCtlArgs.updateParams[i].extMemPtr= (uint8_t*)ivisionBufPlane[i].buf +
(ivisionBufPlane[i].frameROI.topLeft.y * ivisionBufPlane[i].width) +
ivisionBufPlane[i].frameROI.topLeft.x;
        autoIncCtlArgs.updateParams[i].extMemPtrStride = ivisionBufPlane[i].width;
    }

    /* Here we pass createParams->numLevels+1 as the index of the SINK node. We don't
use SINK_NODE because it is set to 6, which is for 5 level of pyramid
     * and wouldn't work if the graph was reconfigured to fewer levels than 5.
     */
    status= BAM_controlNode(graphHandle, intAlgHandle->numLevels+1, &autoIncCtlArgs);

 Exit:
    return status;
}
```

The graph of this function uses TI's auto-increment DMA node as both source and sink nodes, thus the variable `autoIncCtlArgs` of type `EDMA_UTILS_autoIncrement_updateParams` is declared at the beginning of the function. A pointer to this variable will be directly passed to `BAM_controlNode()`. The structure `EDMA_UTILS_autoIncrement_updateParams` allows the set up of several parameters simultaneously:

```
typedef struct
{
    uint32_t transferType;
    uint32_t updateMask;
EDMA_UTILS_autoIncrement_transferProperties updateParams[NUM_MAX_TRANSFER_AUTOINCREMENT];
}EDMA_UTILS_autoIncrement_updateParams;
```

`Indeed updateMask is created by ORing the following macro synmbols:`

```
typedef enum{
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_ROIWIDTH          = (uint32_t)1U << 0U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_ROIHEGIHT         = (uint32_t)1U << 1U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_BLOCKWIDTH        = (uint32_t)1U << 2U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_BLOCKHEIGHT       = (uint32_t)1U << 3U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTBLKINCREMENTX  = (uint32_t)1U << 4U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTBLKINCREMENTY  = (uint32_t)1U << 5U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_INTBLKINCREMENTX  = (uint32_t)1U << 6U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_INTBLKINCREMENTY  = (uint32_t)1U << 7U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_ROIOFFSET         = (uint32_t)1U << 8U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTR         = (uint32_t)1U << 9U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_INTERMEMPTR       = (uint32_t)1U << 10U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTRSTRIDE   = (uint32_t)1U << 11U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_INTERMEMPTRSTRIDE = (uint32_t)1U << 12U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_DMAQUENO          = (uint32_t)1U << 13U,
 EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_ALL               = 0xFFFFU
}EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_TYPE;
```

In the case of image pyramid, the applet only allows the updates of `EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTR` and `EDMA_UTILS_AUTOINCREMENT_UPDATE_MASK_EXTMEMPTRSTRIDE` parameters. The remaining parameters keep the values that were assigned at graph create time.

In case of image pyramid, the source's `autoIncCtlArgs.updateParams[0].extmemPtr` is calculated differently depending on whether the filtering algorithm is a 5x5 Gaussian or a simple 2x2 averaging. In case of Gaussian filtering, the coordinates of the upper left corner of the ROI are shifted towards the top left of the frame by amount of `extraBorder/2` in order to include the border pixels required during filtering.

The sink node has `intAlgHandle->numLevels` outputs and each output's `extmemPtr` and `extMemPtrStride` are getting updated with the information passed through `outBufDesc`.

8.2.2 Changes to <APPLET>_TI_process()

This function is implemented in file *<APPLET>_alg.c*. The changes involve calling `<APPLET>_TI_dmaControl()` to update the transfer parameters and getting rid of the input arguments of type `BAM_InBufs` and `BAM_InBufs` when calling `BAM_process()`. The following shows the changes made to `IMAGE_PYRAMID_U8_TI_process ()` from BAM 2.02 to BAM 2.03. Highlighted in red is old code, which is deleted in the new version and in green is new code.

```
int32_t IMAGE_PYRAMID_U8_TI_process(IVISION_Handle Handle,
        IVISION_InBufs *inBufs,
        IVISION_OutBufs *outBufs,
        IVISION_InArgs *inArgs,
        IVISION_OutArgs *outArgs)
{
    IMAGE_PYRAMID_U8_TI_Handle algHandle = (IMAGE_PYRAMID_U8_TI_Handle)(Handle);
    IMAGE_PYRAMID_U8_TI_Handle intAlgHandle;
    int32_t status            = IALG_EOK;
    int32_t i, j;

    uint16_t extraBorders;

    IBAMALG_InArgs          bamInArgs;
    IBAMALG_InBufs          bamInBufs;
    IBAMALG_BufDesc         inBufDesc[1];

    IBAMALG_OutArgs         bamOutArgs;
    IBAMALG_OutBufs         bamOutBufs;
    IBAMALG_BufDesc         outBufDesc[IMGPYRAMID_MAX_NUM_LEVELS];

    IMAGE_PYRAMID_U8_TI_outArgs * pyramidOutArgs;

    /*-----------------------------------------------------------------------*/
    /* Activate the algorithm to make sure that now onwards internal memory  */
    /* handle can be utilized                                                */
    /*-----------------------------------------------------------------------*/
    IMAGE_PYRAMID_U8_TI_activate((IALG_Handle)Handle);

    intAlgHandle                      = (IMAGE_PYRAMID_U8_TI_Handle)algHandle-
>memRec[ALG_HANDLE_INT_MEMREC].base;
    intAlgHandle->graphMem.graphObj      = intAlgHandle->memRec[GRAPH_OBJ_INT_MEMREC].base;

    /*-----------------------------------------------------------------------*/
    /* BAM specific In Args and OutArgs Assignment                           */
    /*-----------------------------------------------------------------------*/
    bamInArgs.size                 = sizeof(IBAMALG_InArgs);
    bamInArgs.sliceIndex           = 0;

    bamOutArgs.size                = sizeof(IBAMALG_OutArgs);
#ifdef _OLD_DMA_CTL
    bamInBufs.numBufs              = 1;
    bamInBufs.bufDesc              = inBufDesc;

    bamOutBufs.numBufs        = intAlgHandle->numLevels;
    bamOutBufs.bufDesc              = outBufDesc;

    extraBorders= 0;
    for (i=0;i<intAlgHandle->numLevels;i++){
        extraBorders+= (1<<i)*4;
    }
#endif
    BAM_activateGraph(intAlgHandle->graphMem.graphObj);

    /*--------------------------------------------------------------------*/
    /* Loop to support N Frame in single call from user                   */
    /*--------------------------------------------------------------------*/
    for(j = 0; j < inBufs->numBufs; j++)
    {
#ifdef _OLD_DMA_CTL
        BuffDescTranslation(&(bamInBufs.bufDesc[0]), &(inBufs->bufDesc[j]->bufPlanes[0]));

        /* In case gaussian filter is used, we rewind offset.x and offset.y by the exact border
width and height required for a 5x5 gaussian
         * This is to ensure that the pointer for EDMA read points exactly to the upper left
corner of the region composed
         * by the ROI augmented by the filtering border.
         * Also bufSize.width and bufSize.height need to be adjusted to include the border pixels
         * */
        if (intAlgHandle->filterType== IMAGE_PYRAMID_U8_TI_5x5_GAUSSIAN) {
            bamInBufs.bufDesc[0].sliceOffset.x   -= extraBorders/2;
            bamInBufs.bufDesc[0].sliceOffset.y   -= extraBorders/2;
            bamInBufs.bufDesc[0].bufSize.width   += extraBorders;
            bamInBufs.bufDesc[0].bufSize.height  += extraBorders;
        }

        for (i=0 ; i<intAlgHandle->numLevels; i++) {
            BuffDescTranslation(&(bamOutBufs.bufDesc[i]),&(outBufs->bufDesc[j]->bufPlanes[i]));
        }

#else
        status= IMAGE_PYRAMID_U8_TI_dmaControl(&(intAlgHandle->graphMem), intAlgHandle, inBufs-
>bufDesc[j], outBufs->bufDesc[j]);

        if (status!= IALG_EOK) {
            goto Exit;
        }
#endif

        /*--------------------------------------------------------------------*/
        /* Call execute function                                              */
        /*--------------------------------------------------------------------*/
#ifdef _OLD_DMA_CTL
        status =  IMAGE_PYRAMID_U8_TI_execute(&(intAlgHandle->graphMem), &bamInBufs, &bamOutBufs,
&bamInArgs, &bamOutArgs);
#else
        status =  IMAGE_PYRAMID_U8_TI_execute(&(intAlgHandle->graphMem), &bamInArgs, &bamOutArgs);
#endif
        /*--------------------------------------------------------------------*/
        /* iVISION doesn't allow multiple control output, so last utput will be  */
        /* available to user                                                  */
        /*--------------------------------------------------------------------*/
        if (outArgs->size == sizeof (IMAGE_PYRAMID_U8_TI_outArgs) )
        {
            pyramidOutArgs = (IMAGE_PYRAMID_U8_TI_outArgs *)outArgs;
            pyramidOutArgs->outputBlockWidth= intAlgHandle->outputBlockWidth;
            pyramidOutArgs->outputBlockHeight= intAlgHandle->outputBlockHeight;
            pyramidOutArgs->activeImgWidth= intAlgHandle->activeImgWidth;
            pyramidOutArgs->activeImgHeight= intAlgHandle->activeImgHeight;
        }
    }

Exit:
    /* Below BAM_deActivateGraph() is commented out as there is no need to save the context after
each BAM_process() because this particular evelib_imagePyramid_u8() function
     * doesn't change any of its internal context. The context always stays in the same state that
was found after BAM_createGraph()
     * */
    /*
    BAM_deActivateGraph(intAlgHandle->graphMem.graphObj);
     */
    return status;
}
```

We see that the implementation of `IMAGE_PYRAMID_U8_TI_process()` has become simpler as a lot of the old code dealing with translating DMA transfer parameters has moved into `IMAGE_PYRAMID_U8_TI_dmaControl()`. This makes the implementation cleaner because these translations are mostly applet dependent and it makes more sense to put them in file *<APPLET>_graph.c.*

Also the call to `IMAGE_PYRAMID_U8_TI_execute()` doesn't include the arguments bamInBufs and bamOutBufs anymore.