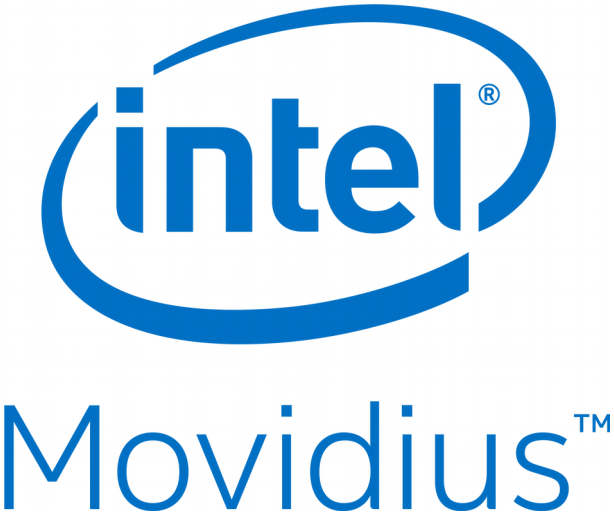


Myriad 2 Development Kit (MDK)

Programmer's Guide

v4.62 / August 2018

Copyright and Proprietary Information Notice

Intel® Movidius™
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Revision History

Date	Version	Description
August 2018	4.62	Updated `bdpart` and `fstab` links. Corrected typo: The Shave L2 cache line is 64 bytes.
June 2018	4.61	Updated section 5.4.3, Tweaking build options with new examples.
March 2018	4.6	Updated section 2.3.3, Cache Coherency with regard to LEON aspects that programmers need to be cautious about.
February 2018	4.5	Updated table in section 7.3, RTEMS in MDK. Added sections 7.6.3, Automated RTEMS build and 7.6.4, Manual RTEMS build replacing sections “Installations steps”, “Using the new libraries”, “More details about customized RTEMS builds”. Updated section 7.7, RTEMS benchmark suite. Updated links in chapter 12, References. Added section 3.4, Shave application stack instrumentation on new shave stack instrumentation option via moviCompile. Added shave stack overflow check information for MISA FW in step 6, Configure stack instrumentation for shave applications and added steps 7, 8 in section 8.7.3, Basic Build Configuration.
December 2017	4.4	Updated section 8.7.3 Basic Build Configuration: removed steps 6 and 7.
November 2017	4.3	Updated section 8.7.2.1.2 OsDrvSvuOpenShaves. Updated section 8.7.2.1.10 SwcRequestUnallocatedShaves / OsDrvRequestUnallocatedShaves. Updated section 8.7.3 Basic Build Configuration. Updated section 8.7.5.1 Example module data configuration. Renamed and updated section 8.8 Runtime instantiated applications. Renamed `SHAVE_APP_ENABLE_DEBUG_CONTEXT` to `DEBUGLOADCONTEXT`.
August 2017	4.21	Fixed references to eFUSE Programming Application Note.

Table of Contents

1 Introduction

1.1 Overview

This guide describes the functionality and use of the Myriad 2 multiprocessor SoC family from a programmer's perspective.

This chapter provides a general overview of the Myriad 2 devices and also introduces some other relevant documentation.

1.2 Myriad 2 family overview

The Myriad 2 SoC device family offers twelve SHAVE vector processors with two 32-bit RISC (LEON), a host of on-chip hardware accelerators and IO peripheral interfaces to provide exceptional performance efficiency and flexibility. A brief overview of the Myriad 2 common features are presented below:

- 12 x SHAVE VLIW vector processor, 2 x RISC processor.
- There is 2 MB of on-chip RAM (CMX).
- 128/512 MB of in-package stacked DDR.
- LEON RISC has 256 KB L2 cache memory.
- LEON RT has 32 KB L2 cache memory.
- Exceptionally high sustainable on-chip bandwidth.
- SIPP Image Signal Processing hardware accelerators.
- Wide range of IO peripherals interfaces, such as SPI (3), I2C (3), I2S (3), SDIO, Ethernet, USB.
- Imaging interfaces, such as MIPI (6), CIF (2), LCD.

The Myriad 2 family consists of the following socket revisions:

- MA2x5x – MA2150 / MA2155 / MA2450B / MA2450C / MA2455B/ MA2455C.

1.2.1 The Myriad 2 MA2x5x series

MA2x5x is the second series of the Myriad 2 family with four members (MA2150 / MA2155 / MA2450B / MA2450C / MA2455B/ MA2455C) and the following specific features:

- 600 MHz system clock.
- USB 3.0 operation.
- Dual voltage SDIO 1.8V & 3.3V.
- USB boot mode.
- IQ and performance improvements in SIPP ISP pipeline – 600 MPixels/s throughput.
- On-die temperature sensors.
- 6 simultaneous camera support.
- 128MB of in-package stacked LP-DDR2 @ 533MHz (MA215x).
- 512MB of in-package stacked LP-DDR3 @ 732MHz (MA245xB).
- 512MB of in-package stacked LP-DDR3 @ 912MHz (MA245xC).
- Secure boot mode (MA2x55).
- Low power state improvements.

The MA2150 architecture block diagram can be found on Figure 1.

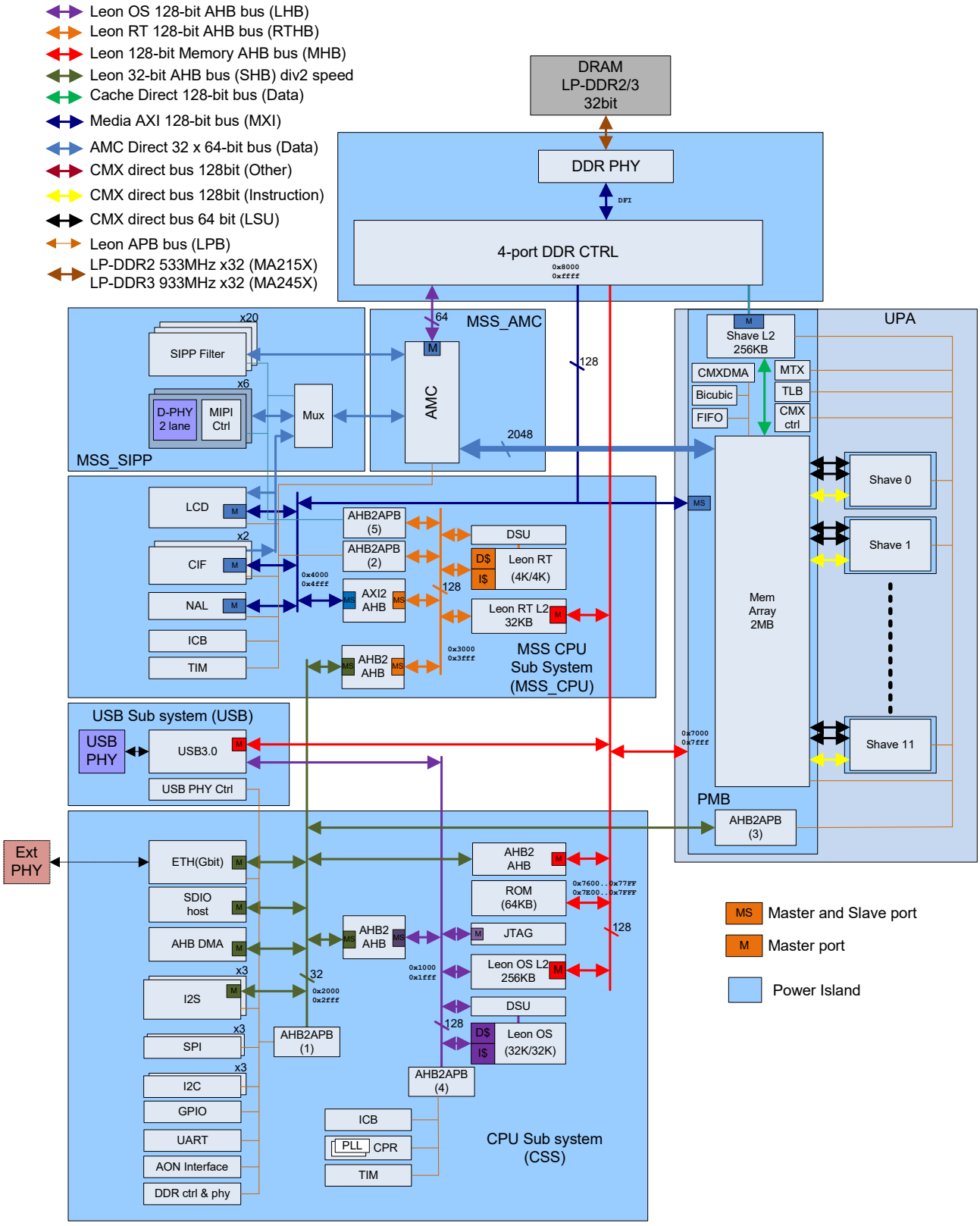


Figure 1: MA2150 Architecture Diagram – Block level

1.3 Myriad 2 Block level architecture overview

The block diagrams presented in the previous chapter show the three major architectural units of the Myriad 2 processor: The Media Sub System (MSS), the CPU Sub system (CSS) and the Microprocessor Array (UPA).

1.3.1 The Media Sub System (MSS)

The MSS is the architectural unit designed to allow connections with imaging devices (camera sensors, Display devices, HDMI controllers etc.) as well as allowing use of the SIPP HW filters. The MMS processing flows are comprised of the MIPI, LCD, CIF IO interfaces, the SIPP HW filters and the AMC block which enables connections between these and CMX (SRAM) memory.

The Leon RT RISC coordinates frame input and the MSS processing pipelines. The Leon RT is only Interface or HW filter register settings so it can efficiently change any required parameters of the MSS blocks with the minimum amount of delay due to bus arbitration.

1.3.2 The CPU Sub System (CSS)

The CSS have been designed to be the main communication and control unit with the outside world via the external communication peripherals: I2C blocks, I2S blocks, SPI blocks, UART, GPIO, ETH and USB3.0. The Leon OS (LOS) RISC is the control unit of this block. The Leon OS has the capability to run RTOS due to larger L1 (256 KB) and L2 (256 KB) caches. The CSS block also offers an AHB DMA engine for more optimal data transfer via the external peripherals. Beside handling the external interfaces the Leon OS typically also controls software running on the SHAVE processors.

1.3.3 The Microprocessor Array (UPA)

The microprocessor array (UPA) contains the 16 VLIW SHAVE vector processors with shared 2 MB CMX SRAM memory. In addition, it contains a specialized DMA engine, and 256 KB of L2 cache memory available to the SHAVE vector processors.

This UPAs main purpose is to provide support for VLIW optimized code required by many imaging or computer vision application's as well as any other general computation intensive algorithms.

The CMX memory itself will be described in greater detail in a further chapter, but one aspect should be highlighted at this stage: each SHAVE processor has preferential ports into a 128 KB slice of the CMX memory. As such, 16x128 KB are preferentially used by SHAVE cores but the remaining 512 KB of CMX memory are generally usable by any other resources. The recommended usage for these 512 KB is for HW SIPP filters usage or Leon OS timing critical code which would otherwise not be able to be kept in DDR.

1.4 Myriad 2 programming paradigms

1.4.1 Standard programming paradigm

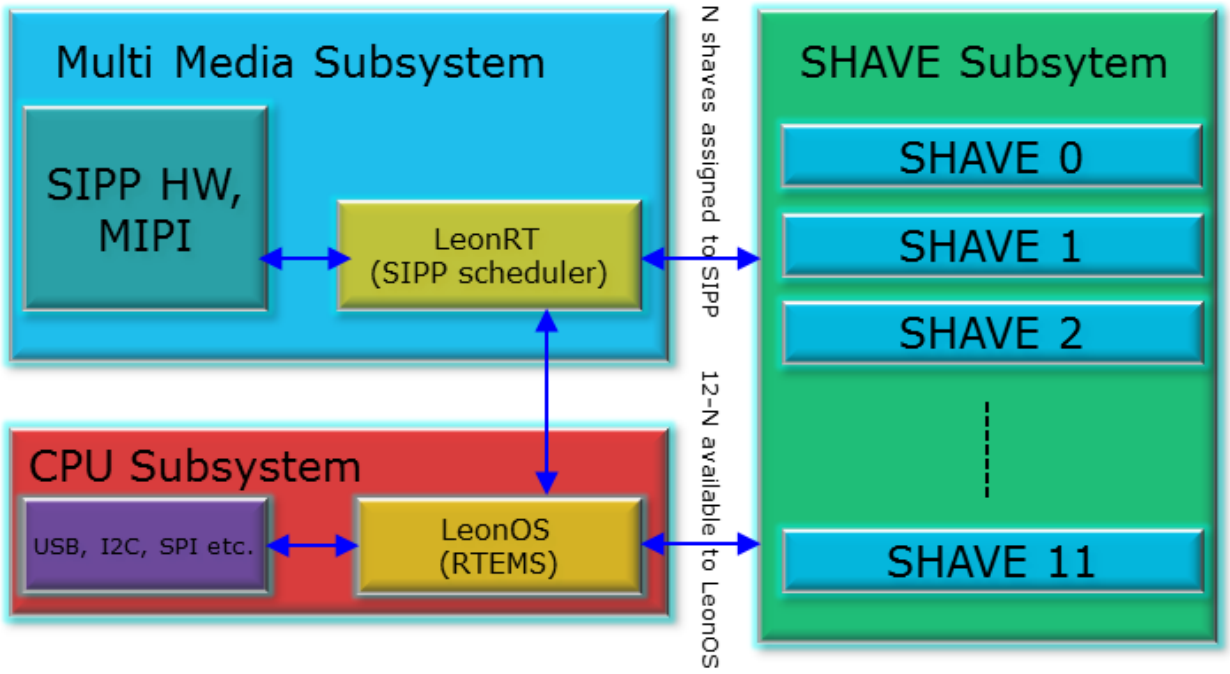


Figure 2: Standard programming paradigm

The standard programming paradigm for Myriad 2 involves:

- Using RTEMS on BOTH LeonOS and LeonRT.
- The SIPP scheduler running on LeonRT (with RTEMS).

The advantage of this paradigm is that it provides parallelization in an easy to use environment. The SIPP scheduler itself is able to ensure parallel pipeline configurations for managing the MSS HW filters and external interfaces with a low footprint so as to ensure optimal LeonRT utilization.

The SIPP pipelines can also use SHAVEs software kernels. Any remaining SHAVEs not used for SIPP pipelines are remain free to be used by the Leon OS for application specific software tasks including (but not limited to) computer vision algorithms.

1.4.2 The One Leon programming paradigm

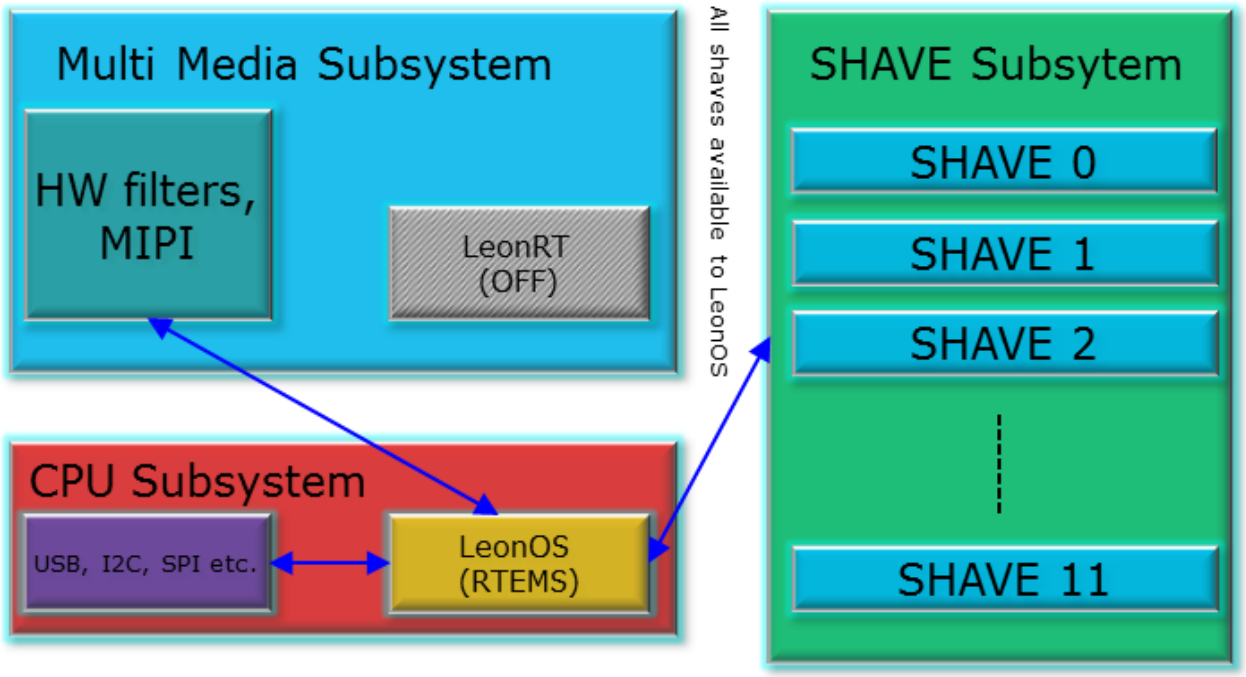


Figure 3: One Leon programming paradigm

Some applications might not require heavy line based processing. Such applications might choose to completely switch OFF the Leon RT processor and instead only use LeonOS with (or without) RTEMS. HW filters may still be used. Using this programming paradigm, Leon OS would control all of the applications running on the 12 SHAVE cores.

1.4.3 Bare metal programming paradigm

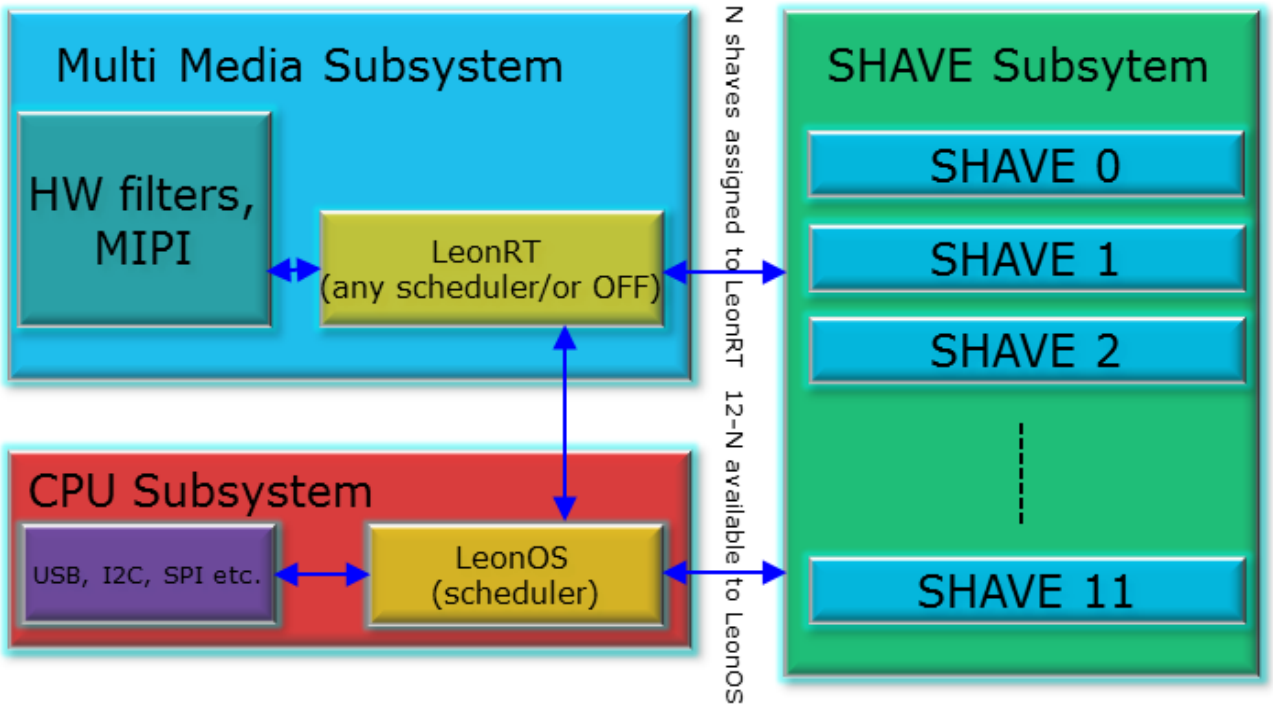


Figure 4: Bare metal programming paradigm

A bare metal programming paradigm will also be supported by the MDK build system. This will allow developer to use both LEON cores without any operating system, only minimal schedulers running to control the pipelines application.

This paradigm requires more integration efforts but allows developers to write applications which will not be affected by any operating system overhead.

1.5 Other relevant documentation

Supporting documentation is available in addition to this guide and is referenced in chapter 12. These documents are referenced in this guide as [refX], where relevant.

The scope of this document is to provide an overview of all elements of the Myriad platform from a programmer's perspective, and further detail is generally available in the more focused reference material.

2 LEON

2.1 Introduction

LEON4 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption.

The LEON4 core has the following main features: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider.

The SPARC Architecture Manual provides an excellent description of this processor, see [ref1]. Further important reading about caches and Leon is the Leon IP core manual [ref2].

2.2 Overview

There are 2 LEON4 cores used. They will be referred to in this document as LeonOS and LeonRT. Their purposes/usage was described in the introductory chapter. The LEON4 core features:

- SPARC V8 instruction set with V8e extensions.
- Advanced 7-stage pipeline.
- Hardware multiply and divide units.
- High-performance, fully pipelined IEEE-754 FPU.
- Separate instruction and data cache (Harvard architecture) with snooping.
- Instruction and data caches.
- AMBA-2.0 AHB bus interface.
- Advanced on-chip debug support with instruction and data trace buffer.
- Power-down mode.
- Reference MMU implementation.
- Single vector trapping enabled.
- Multi-vector trapping selectable.

2.3 Programming details

2.3.1 Addressing Scope

Both LeonOS and LeonRT processors have complete access to the flat Myriad 32-bit address space. But it is important to consider that LeonOS is placed in the CSS and LeonRT in the MSS. As such, fewer bridges are crossed if LeonRT makes accesses to MSS peripherals and LeonOS to CSS peripherals.

2.3.2 Booting

Upon power-up, or after reset, LeonOS starts executing from its internal ROM memory. The firmware therein allows the loading of application code and data from an onboard FLASH chip or from an application processor connected to the SPI bus. The loader has full access to all the on-chip memories, allowing it to load any application. Upon completion of the loading process, the boot loader passes control to the loaded application's `LEON` entry point, which does some basic initialization before calling the `main()` function or the `POSIX_Init()` function. In this way, Myriad supports running any production code, as any built application can be converted to the binary format used for boot-time loading.

`LeonOS` is the sole bootable processor on Myriad and serves as the entry point of the entire Myriad system. All applications running on Myriad without a debugger attached need to have a `LeonOS main()` or `POSIX_Init()` function defined, even if its only purpose is to just start one of the `SHAVEs` or `LeonRT`.

Booting is supported over many different interfaces, SPI, i2c, EBI etc. Further details are available in the platform architecture document [ref3].

2.3.3 Cache Coherency

Separate instruction and data caches are available in the LEON subsystem, which significantly improves LEON performance. The data cache is a write-through cache memory. The nature of this cache requires that special care must be taken if data is shared between LEONs and SHAVEs. Macros are provided in the swcLeonUtils library to facilitate reads by LEONs on data shared and updated by SHAVEs or between Leon processors themselves – L1 cache bypass reads are required on the Leon when accessing memory from another core than itself. Using the "volatile" qualifier on this data is insufficient. Please refer to Figure 6 for a diagram about the cache organization in Myriad 2.

The Leon L2 caches may be configured to only cache parts of the memory.

The memory hierarchy architecture in Myriad 2 does not guarantee data coherency between accesses to cacheable and non-cacheable memory locations. This is a consequence of concurrent data paths between Leon's integer unit (core) and other peripherals.

Sequences of alternating transfers to cacheable and non-cacheable memory location will always lead to race conditions.

To solve this coherence aspect, one must employ an iterative approach (depending on which caches are active) and avoid the un-cacheable read-check mechanism in the following manner:

- For L1: this is always in write-through mode, so a write-read-check sequence must issue a cache invalidation operation after the last write before issuing the read-check.
- For L2: either employ the same mechanism as for L1 if the cache is in write-back or, preferably, maintain the cache in copy-back mode and issue a cache flush at the end of the last write sequence.

2.3.4 LEON tools

LEON code compilation and linking are facilitated by the gcc and binutils packages. The ELF format is used for object and executable file. Gcc and binutils are available together in the form of a sparc-elf-gcc package. These tools are integrated in the MDK build flow alongside the Myriad specific tools.

2.3.5 Endianness

The Leon processors are both working in little endian same as the SHAVE processors. No extra requirements are needed to ensure endianess consistency.

2.3.6 Interrupt Handling

The interrupt handling entry points are hand-written in the SPARC assembler, and thoroughly optimized, ensuring that the interrupt handling overhead is as low as possible, thus maximizing the available cycles for the user-implemented Interrupt Service Routines. This approach allows implementations of both synchronous and asynchronous application paradigms.

Note that floating point register are not correctly saved and restored by the interrupt service routine. This means that it is unsafe to use floating point data types in Leon ISRs.

Myriad 2 LEONs support both single vector traps and multi-vector traps but in most cases the developer does not need to be aware which one of these is enabled as the components/drivers interfaces will account for proper usage in the background themselves.

Both LeonOS and LeonRT have their own Interrupt Controller Block (ICB) circuitry.

2.3.7 Timers

Each LEON processor has its own block of timers.

There are eight general purpose timers, a free running counter, a random number generator and a watchdog timer. There is a facility to pre-scale the system clock to allow timers to run at slower speeds. Each general purpose timer can generate an individual interrupt to its Leon core interrupt controller.

2.3.8 Ancillary State Register 17 (ASR 17)

The Leon IP documentation, [ref2] lists the existence of different registers. Of particular interest is ASR17 because of its role for characterizing a LEON processor. In the case of the Myriad architecture, here is how they are used:

Field name	Bit(s) position	Description
Processor index (PI)	[31:28]	In multi-processor systems, each LEON core gets a unique index to support enumeration. The value in this field is identical to the hindex generic parameter in the VHDL model if smp = 1, or from the irqi.index signal if smp = 16.
Clock switching enabled (CS)	[17]	If set switching between AHB and CPU frequency is available.
CPU clock frequency (CF)	[16:15]	CPU core runs at (CF+1) times AHB frequency.
Disable write error trap (DWT)	[14]	When set, a write error trap (tt = 0x2b) will be ignored. Set to zero after reset.
Single-vector trapping enable (SVT)	[13]	If set, will enable single-vector trapping. Fixed to zero if SVT is not implemented. Set to zero after reset.
Load delay	[12]	If set, the pipeline uses a 2-cycle load delay. Otherwise, a 1-cycle load delay i s used. Generated from the lddel generic parameter in the VHDL model.
FPU option	[11:10]	“00” = no FPU; “01” = GRFPU; “11” = GRFPU-Lite
	[9]	If set, the optional multiply-accumulate (MAC) instruction is available
	[8]	If set, the SPARC V8 multiply and divide instructions are available.
	[7:5]	Number of implemented watchpoints (0-4)
	[4:0]	Number of implemented registers windows

The ASR17 register, on Myriad, at boot time, will resolve into these two values:

- LeonOS ASR17 = 0x00004587
- LeonRT ASR17 = 0x10004587

Both have the same values but the processor index bits may be used in software to determine which one of the Leon processors we are currently running on this is why the LeonRT is OR’ed with 0x10000000.

After boot, the crt0.S initialization file will change the value in these registers to:

- LeonOS ASR17 = 0x00006587

- LeonRT ASR17 = 0x10006587

Enabling SVT and DWT. Please remember any of the following values are also possible considering that bits 13 and 14 are writable:

- LeonOS ASR17 = 0x00000587, 0x00002587
- LeonRT ASR17 = 0x10000587, 0x10002587

2.4 Typical LeonOS usage

The Movidius Development Kit features driver functions and low level components that enable the user to use the `LeonOS` processor as a control processor for peripherals and `SHAVE`s.

Despite `LeonOS` being capable of numeric computation, its capabilities are significantly inferior to those of the `SHAVE`s. For this reason, `LeonOS` is most generally used for controlling the peripherals, data flow control and also for application state management, while all computation tends to be implemented on `SHAVE`s.

2.4.1 Special memory features of the OS Leon

The OS Leon has a 256 KB L2 cache memory which makes it suitable for running small operating systems even if the code of these resides in DDR memory.

The L1 caches are 32 KB each (for instruction and data) which speeds up `LeonOS` even further.

`LeonOS` may execute from any of the system memory areas: `CMX` or `DDR`. The system design is specifically tailored to allow for running from DDR with minimum penalty due to optimally chosen cache sizes.

2.4.2 Special memory features of the RT Leon

The RT Leon has 4 KB L1 instruction and data caches and a 32 KB L2 cache memory.

2.5 ISO C/C++ language library support

As stated previously, the nature of the `LEON` goals is to control the various peripherals, interrupts and state machine events. The primary initial focus of the `LEON` development environment was directed towards reliable, fast, booting and trap handling code.

ISO C/C++ is supported on `LEON` processors on both bare-metal and RTEMS OS environments with the following limitations:

- No console support.
- No file system support.
- No exception handling.

The C std libraries libraries are implemented using newlib/libgloss.

2.5.1 heap

The default heap is set to 6KB, but a different larger heap may be used if required by configuring a separate memory buffer in the application and set it as a heap with the `mvSetHeap` function.

2.5.2 printf

The output of function `printf` is sent to UART. If used on a system that has a debugger connected, `printf` messages are shown in the `moviDebug` window.

The developers must take care not to leave any stray `printf` functions in the code that should run without a JTAG cable connected. When this happens, the application just hangs indefinitely.

It is important to know that the Myriad processor only flushes its UART queue over JTAG when a "`\n`" character is received. Developers must ensure that they use this character in at least one of the `printf` commands from their application.

3 SHAVE

3.1 SHAVE overview

SHAVE contains wide and deep register files coupled with a Very Long Instruction Word (VLIW) for code-size efficiency. As shown below, VLIW packets control multiple functional units which have SIMD capability for high parallelism and throughput at a functional unit and processor level. Each of these units can be launched in parallel in a single instruction packet.

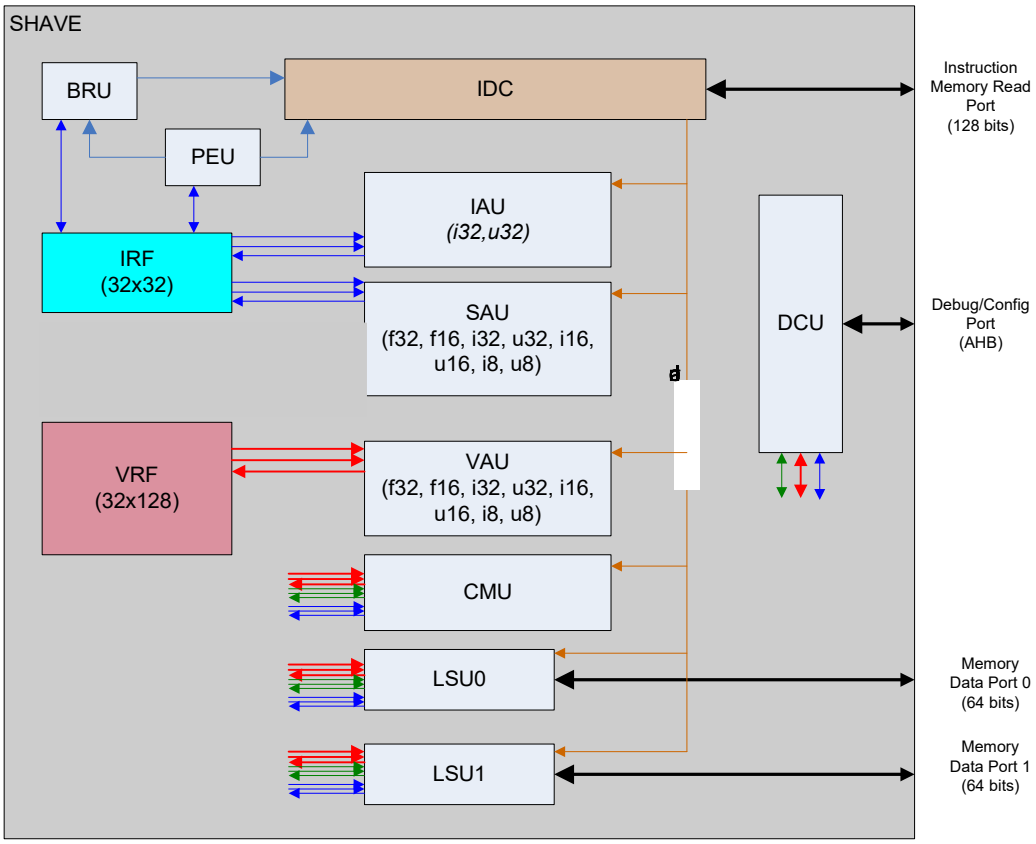


Figure 5: SHAVE Internal Architecture

SHAVE operates in little endian mode.

SHAVE supports SIMD instructions on multiple types, including but not limited to: 16 bits integer, 32 bits integer, 16 bits float, 32 bits float, 8 bits integers.

There is Assembly, C and C++ support for programming the SHAVE through the use of moviAsm and moviCompile, both of which are Movidius internally developed tools.

There are two register file arrays: IRF and VRF which are described shortly below. The VAU, SAU, IAU, PEU, BRU and LSUs are also detailed.

3.1.1 IRF (Integer Register File)

The IRF consists of 32 registers, each 32 bits in length. These registers are implemented mainly to support integer operations, but are also used for load and store instructions.

The execution units operating with these registers are the IAU (Integer Arithmetic Unit), and the SAU (Scalar Arithmetic Unit).

There are SIMD operations operating with 16 bits and 8 bits integer data types performed by the SAU on the IRF.

3.1.2 VRF (Vector Register File)

The VRF consists of 32 registers, each 128 bits in length. The purpose of these registers is to provide SIMD operations to the SHAVE.

The arithmetic unit operating with VRF is the VAU (Vector Arithmetic Unit). This supports both integer and floating point operations directed towards multiple data types: 8, 16, 32-bit data types of integer or floating point.

3.1.3 IAU (Integer Arithmetic Unit)

This unit provides integer operation support on the IRF registers as well as support for different shifting and logic operations.

3.1.4 SAU (Scalar Arithmetic Unit)

This unit provides floating point operations support on the IRF.

Besides the most common floating point operations, this unit implements a few more complex functions on 16 bits floating point including: reciprocal, sine, square root, reciprocal of square root, cosine, arctangent, logarithm and exponential.

The unit also provides integer operations on the IRF registers. This feature may be used to launch more integer operations in parallel on the IRF if found useful.

3.1.5 VAU (Vector Arithmetic Unit)

This unit provides both floating point and integer operations on the VRF registers using 8, 16, and 32-bit data types of both integer or floating point.

3.1.6 CMU (Compare and Move Unit)

This unit provides functionality to copy (move) data from one register file to the other. Any combination is possible and multiple bit lengths are supported.

The unit also provides functionality for comparing different data types. Comparison is done setting a Condition Code register with multiple entries. This allows for comparisons to be made on VRF registers too, comparing multiple data at once.

3.1.7 LSU (Load Store Unit)

There are two load store units which provide functionality for loading and storing data to both register files.

The LSU, used in conjunction with other units can also provide swizzling operations on various data types as described in the SHAVE ISA document.

3.1.8 BRU (Branching Unit)

The BRU provides functionality for branching. The SHAVE has a delay slot of 6 cycles which may be used to fill in other instructions.

3.1.9 PEU (Predicate Execution Unit)

The PEU is helpful for implementing conditional branching and also to make conditional stores on LSU or VAU units.

3.2 Running SHAVE code

3.2.1 Instruction code considerations

The SHAVE processor has access to both L1 and L2 cache memories. L1 cache memory is organized is both instruction and data cache.

When designing the SHAVE cache architecture, several Myriad 1 algorithms were evaluated and as a result of this evaluation the L1 instruction cache was chosen to be 2 KB in size. This allows running SHAVE code from DDR without paying a high penalty for doing so.

3.2.2 Data usage considerations

The SHAVE processor features also a 1 KB data L1 cache.

The Shave can also make use of a 256 KB L2 cache for data caching. The Shave L2 cache may be configured in up to 16 partitions of 16 KB each. The LSU units may be assigned any partition afterward.

The Shave L2 cache line is 64 bytes.

A write-back policy is implemented by default, meaning that write data is typically written to the cache instead of being written directly to memory. This cache line is flagged as *dirty* and written to external memory if:

- The cache line is *flushed* by the shave.
- The entire cache is *flushed* by the shave.
- A read or write request yields a cache miss but is targeted at a cache entry that is dirty.

A write-through policy may be enforced per transaction by asserting the signal *sve_writethru*. In this case all write requests write to external memory as well as the cache. Dirty bits are never set in the tag memory in this case.

3.3 Standard Library support

3.3.1 Standard types

The Movidius C/C++ compiler comes with support for some vector types which are enumerated below. These may be accessed through the use of `moviVectorUtils.h`. These are:

- int4
- uint4
- 3x32-bit
  - int3
  - uint3
- 2x32-bit
  - int2
  - uint2
  - float2
- 8x16-bit
  - short8
  - ushort8
  - half8
- 4x16-bit
  - short4
  - ushort4
  - half4
- 2x16-bit
  - short2
  - ushort2
  - half2
- 16x8-bit
  - char16
  - uchar16
- 8x8-bit
  - char8
  - uchar8
- 4x8-bit
  - char4
  - uchar4
- 2x8-bit
  - char2
  - uchar2

3.3.2 Standard C libraries

The other standard libraries delivered with `moviCompile` are:

- `limits.h` contains integer types limits.
- `math.h` contains function declarations for basic mathematical operations.
- `stdarg.h` original file for `GCC`.
- `stdbool.h` contains the definition of the `bool` type.
- `stddef.h` contains the definition of `NULL`, `size_t`, `wchar_t` and `offsetof`.
- `stdint.h` contains integer types definitions and limits.
- `stdio.h` contains `printf`, `sprintf`, `puts`, `putchar` declaration.
- `stdlib.h` contains `abort` and `exit` declarations, functions for dynamic memory management, random number generation, integer arithmetic, searching, sorting and converting.
- `string.h` contains function declarations for string handling.
- `types.h` original file from `GNU C` library.
- `assert.h` contains the `assert` macro definition.
- `ctype.h` contains functions that are used to test characters for membership in a particular class of characters.

3.3.3 Standard C++ library support

- `algorithm` defines a collection of functions especially designed to be used on ranges of elements.
- `array` defines fixed-size sequence containers (C++11).
- `bitset` defines the `bitset` class.
- `cassert` defines the `assert` macro.
- `cctype` declares a set of functions to classify and transform individual characters.
- `cerrno` defines the `errno` macro.
- `cfenv`.

- `cfloat` describes the characteristics of floating types.
- `ciso646`.
- `climits` defines constants with the limits of integral types.
- `cmath` declares a set of functions to compute common mathematical operations and transformations.
- `cstdarg` defines macros to access the individual arguments of a list of unnamed arguments whose number and types are not known to the called function.
- `cstdbool` is to add a `bool` type and the `true` and `false` values as macro definitions.
- `cstddef` defines `ptrdiff_t`, `size_t`, `offsetof`, `NULL`.
- `cstdint` defines a set of integral type aliases with specific width requirements, along with macros specifying their limits and macro functions to create values of these types.
- `cstdio` defines a set of functions which are used for input or output operations.
- `cstdlib` defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting.
- `cstring` defines several functions to manipulate C strings and arrays.
- `ctime` contains definitions of functions to get and manipulate date and time information.
- `cwchar` defines several functions to work with C wide strings.
- `cwctype` declares a set of functions to classify and transform individual wide characters.
- `deque` defines the `deque` container class.
- `exception` defines the base class for all standard exceptions thrown by the elements of the standard library: `exception`. It also defines the special exception `bad_exception`.
- `forward_list` defines the `forward_list` container class.
- `functional`.
- `initializer_list` defines a template class with the same name.
- `iterator` defines the `iterator` class.
- `limits` defines elements with the characteristics of numeric types.
- `list` defines the `list` container class.
- `map` defines the `map` and `multimap` container classes.
- `numeric` describes four algorithms specifically designed to operate on numeric sequences that support certain operators.
- `queue` defines the queue and `priority_queue` container adapter classes.
- `ratio` declares the ratio class template and several auxiliary types to operate with them.
- `scoped_allocator`.
- `set` defines the `set` and `multiset` container classes.
- `stack` defines the `stack` container class.
- `stdexcept` defines a set of standard exceptions that both the library and programs can use to report common errors.
- `string` provides the definitions of the `basic_string` class.
- `tuple` defines the `tuple` class.
- `type_traits` defines a series of classes to obtain type information on compile-time.
- `typeindex`.
- `typeinfo` defines types used in conjunction with the operators `typeid` and `dynamic_cast`.
- `unordered_map` defines the `unordered_map` and `unordered_multimap` container classes.

- `unordered_set` defines the `unordered_set` and `unordered_multiset` container classes.
- `utility` defines pair, `make_pair` and `rel_ops`.
- `valarray` declares the `valarray` class.
- `vector` defines the `vector` container class.
- `ext/hash_map` defines the `hash_map` and `hash_multimap` container classes.
- `ext/hash_set` defines the `hash_set` and `hash_multiset` container classes.

3.4 Shave application stack instrumentation

3.4.1 Enabling the stack instrumentation

In order to enable the stack usage and stack overflow checks for a shave application, the MVCCOP build system variable should be appended with the appropriate moviCompile command line argument, according to the available documentation.

This functionality will allow to better estimate a shave application's stack usage in development stage. It is because all the function calls get instrumented, this option will introduce some overhead and could cause undesired behavior in product life. It is therefore suggested to only use it for development investigation purposes done by the system integrator.

3.4.2 Shave application stack instrumentation

This functionality is properly documented in the moviCompile documentation delivered with the MDK tools package, in the chapter named "Stack Instrumentation".

3.5 Mutexes

3.5.1 Overview

With multiple processors, resources may be shared. The mutex block allows one processor to gain exclusive access to a particular resource.

- Eight individual mutexes.
- `SHAVE`s can predicate-stall on mutex availability to avoid a busy-waiting paradigm.
- `LEON` may poll status or enable a mutex interrupt.

Common uses for mutexes are shared read/write data structures, such as linked lists of tasks, or access to a shared resource (e.g. UART as `stdout`).

3.5.2 Recommended usage

- Get mutexes.
- Perform process that requires mutex.
- Release mutex as soon as possible.
- Make sure that the mutexes are released by a `SHAVE` before it ends execution.

3.6 SHAVE Assembler

3.6.1 Overview

The Movidius Assembler (moviAsm) is the Software Component which is responsible for producing output binary files in the specified format, according to the latest ISA specifications. More information about this may be found in the moviAsm documentation.

An .asm file may be created by the user using a text editor, or by some other tool, e.g. compiler.

3.6.2 moviAsm specifics

MoviAsm supports multiple data types by using suffixes to variables. Loading:

```
lsu0.ldil i0, 256 F3 2 || lsu1.ldih i0, 256 F3 2
```

loads the value 0x43800000 to the i0 register

MoviAsm supports an easy-to-use syntax for specifying instructions for multiple execution units in the same cycle. This is done by the use of " || " as shown above.

MoviAsm supports both local and global labels. The way local labels are supported is through the prefixing of a label with .L.

3.6.3 moviAsm and moviCompile

C code can use handwritten optimized assembly code two ways:

- Through the use of inline assembly.
- By calling shave assembly code written in a calling convention compliant way.

A good application that shows C-assembly SHAVE mix is the SharingExample application delivered together with the MDK.

3.7 Typical SHAVE usage

The SHAVE processor is good at performing computational intensive tasks. Typically, data is buffered from DDR to CMX memory chunk by chunk and processed there, and then moved back to DDR if required after processing is complete.

In order to achieve the best usage of the Myriad SHAVE processors, it is recommended to use C level design for control code on SHAVE and use optimized routines for the inner loops of highly intensive computational tasks.

As stated in previous chapters, it is preferable to let the LEON processors handle the various interrupts while the SHAVE is performing its data processing.

3.7.1 Streaming Image Processing Pipeline (SIPP)

Another typical use of SHAVE processors is to assign them to the SIPP engine. SIPP is a proprietary software/hardware mechanism used by the Myriad 2 processor to achieve highly optimized scheduling of imaging and computer vision processing pipelines. SIPP is described in the SIPP User Guide. Please see section 1.5.

4 Memory Overview

4.1 Introduction

4.1.1 Overview

In an embedded system, managing data locality and DDR bandwidth is key for power efficient operation. This chapter aims to guide a Myriad programmer to make the best use of available resources.

4.1.2 Memory areas

Memory	Size	LEON Access Cost	SHAVE access cost
CMX	2 MB	Low	Low
DDR	128 MB (MA215x) 512 MB (MA245x)	High Low when data cache hit	Low for L1 cache hit Moderate when L2 hit High for random access

4.2 Memory Map

4.2.1 Memory Map table (abridged)

Peripheral	Addresses	Size
Camera 1 (CIF)	30FA0000	-
DDR	80000000	128MB (MA2150) 512MB (MA2450)
LCD1	30FC0000	-
UART	20E00000	-
Clock, Power, Reset (CPR)	10F00000	-
GPIO	20E00000	-
SPI1, SPI2, SPI3	20E40000, 20E50000, 20E60000	-
DDR Controller	20E80000	-
I2C1, I2C2	20E900000, 20EA0000	-
I2S1, I2S2, I2S3	20EB0000, 20EC0000, 20ED0000	-
L2 Cache Control	20FD0000	-
SHAVE Control/Status access: SHAVE-0, SHAVE-1, SHAVE-2, SHAVE-3 SHAVE-4, SHAVE-5, SHAVE-6, SHAVE-7 SHAVE-8, SHAVE-9, SHAVE-10, SHAVE-11	 20F00000, 20F10000, 20F20000, 20F30000 20F40000, 20F50000, 20F60000, 20F70000 20F80000, 20F90000, 20FA0000, 20FB0000	-
LEONOS ROM (LROM)	7FF00000	64 KB
CMX	70000000	2MB

Peripheral	Addresses	Size
CMX Control	20FC0000	-
USB	10F20000	-

4.2.2 Memory Map Notes

The memory map is a programmers' overview of the Myriad memory – areas highlighted in green show memory areas that code/data for a particular processor generally reside in.

- The Myriad debugger, `moviDebug2`, supports accessing many memories/registers by name rather than a specific address. For further details, see the debugger documentation (see section 1.5).
- The above table is an abridged copy of the one available in the Myriad Platform Design Databook (see section 1.5)

4.3 Cache Overview

4.3.1 List of System Caches

PU	Type	Size	Associativity	Cache line size	Policy
SHAVE[N]	L1 Instruction	2 KB	2-way	16 bytes	**read-only** cache
SHAVE[N]	L1 Data (default disabled)	1 KB	Directly mapped	16 bytes	**write-back** policy or **write-through** (configurable)
SHAVES	L2	256 KB	2-way, 1-8 partitions	64 bytes	**write-back** policy
LeonOS	L1 Instructions	32 KB	2-way	32 bytes	**read-only** cache
LeonOS	L1 Data	32 KB	2-way	32 bytes	**write-through** cache
LeonOS	L2	256 KB	4-way	64 bytes	**write-through** or **copy-back** (configurable)
LeonRT	L1 Instructions	4 KB	2-way	32 bytes	**read-only** cache
LeonRT	L1 Data	4 KB	2-way	32 bytes	**write-through** cache
LeonRT	L2	32 KB	4-way	64 bytes	**write-through** or **copy-back** (configurable)

4.3.2 Cache diagrams and notes

4.3.2.1 Leon Cache Hierarchy Diagram

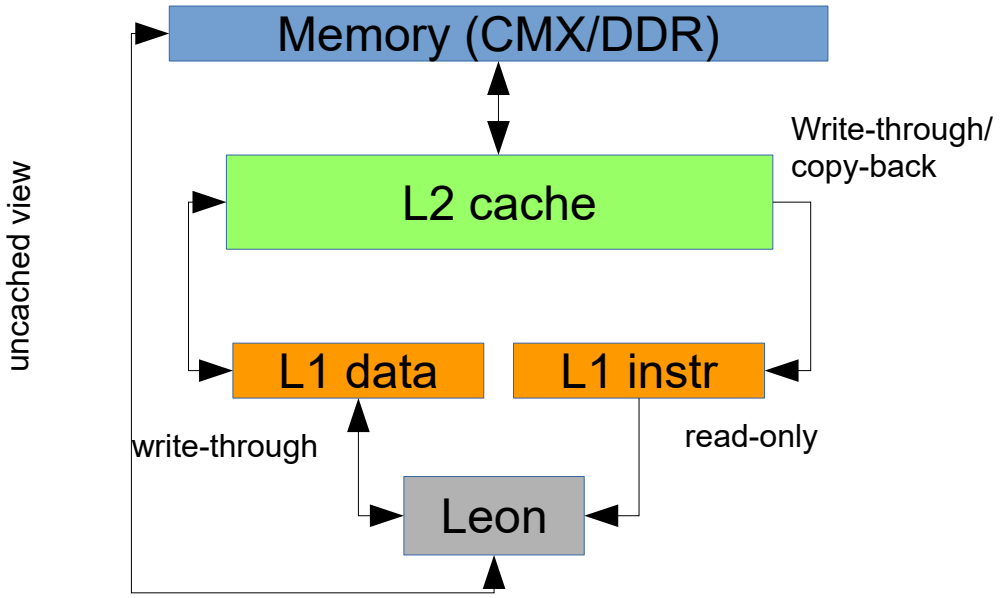


Figure 6: Leon cache hierarchy overview

Figure 6 shows an overview of the Leon cache hierarchy restating the possible configuration options. It is important to take note of a few aspects:

1. L1 instruction cache is read-only. Considering this, L1 instruction cache is generally not required to be handled in any special manner other than activating it at most. This cache is activated by the initialization code in both bare-metal and RTEMS running environments.
2. L1 data cache is a write-through cache. This means that all read transactions will generally happen from the cache, but write transactions will go both in L1 data cache as well as L2 cache. A “flush” operation on this cache will generally not copy data from L1 to L2 cache, but instead only invalidate current blocks.
3. L2 is a cache shared by both L1 data and L1 instruction cache. It can be configured both in write-through, when it caches only reads, writes go in both cache and memory, or in copy-back mode when both reads and writes are cached. In the case when L2 is configured as a copy-back cache, the only way of getting data into memory is by issuing driver level cache flushes. A flush operation for L2 can be configured to do both write back as well as write back and invalidation of the cache at the same time
4. Important to notice that there is also a possibility of accessing data from memory in an uncached view. Both the DDR and the CMX memory have such uncached views. In the case of DDR, addresses having MSB '0' for example: 0x8xxxxxxx represent cached views while addresses starting with MSB '1', for example 0xCxxxxxxx represent uncached views. For CMX, 0x78xxxxxx represent uncached views and 0x70xxxxxx represent cached views. This feature allows easy sharing of control data between the Leon processors and Leons and SHAVE. As stated earlier, if relaying on cache active/bypassed for getting data into memory, different problems could be encountered on account of the cache bits not being touched while in bypass modes. Instead, control structures/arrays may be placed in the uncached views ensuring complete cache bypass by the Leons. For example, one could declare an uncached control variable like:

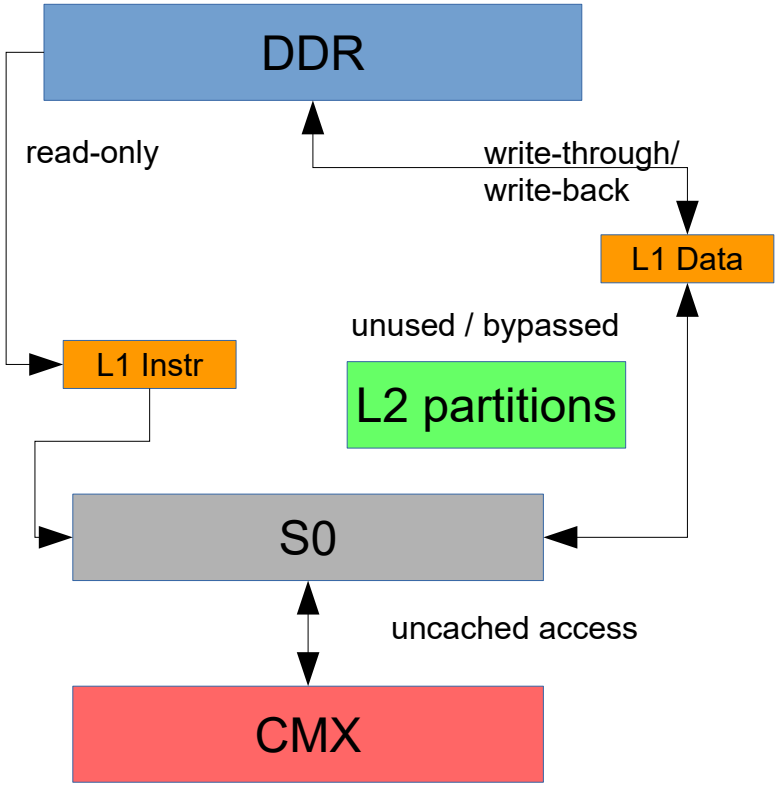
```
volatile unsigned int __attribute__((section(".cmx_direct.data")))
TriggerAct;
```

5. And because it is in an uncached view from declaration time, it will always be treated as uncached by any Leon. This particular feature, although very powerful does come at the expense of slower access to and from the memory when working with these variables. As such, it is highly recommended to use these uncached view variables strictly for small control structures.

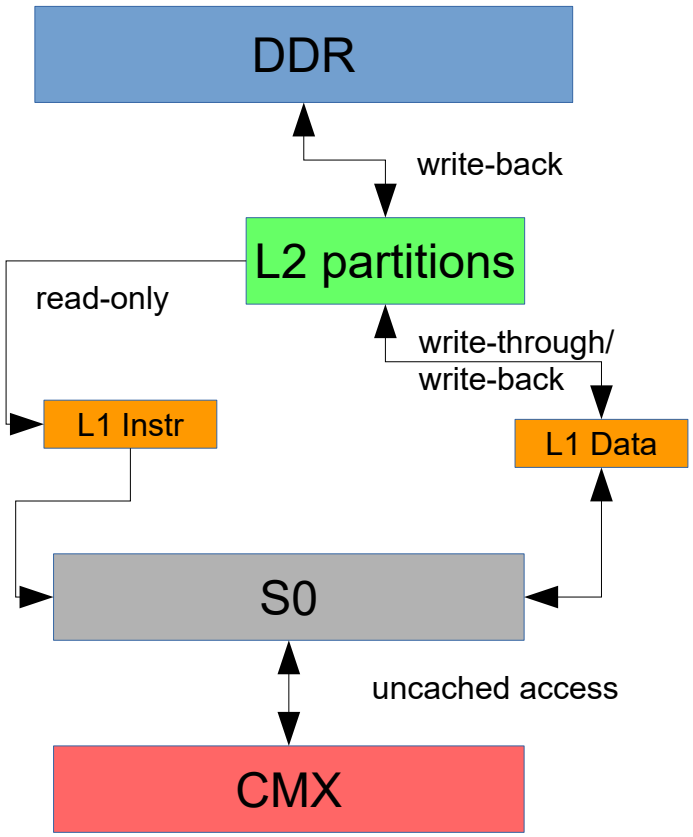
4.3.2.2 SHAVE cache hierarchy diagram

The SHAVE processor will always access CMX memory in an uncached way and it can access DDR memory either directly or through L2 cache partitions. As such, there are two possible configurations:

1. Bypassing L2 cache:



2. Going through the L2 cache:



Considering the structure of memory accesses via the cache infrastructure the following points are drawn to attention:

1. CMX memory is uncached. This makes it the perfect candidate for sharing control structures between SHAVEs or SHAVEs and Leon processors. An important note here is that SHAVEs can access using both cached and uncached views of CMX 0x78xxxxxx and 0x70xxxxxx. This means the similar declared variables:

```
volatile unsigned int __attribute__((section(".cmx_direct.data")))
TriggerAct;
```

2. May be shared for control code sharing between SHAVEs and Leons. The same note made for Leons is valid here: it is highly recommended to use strictly smaller structures only for control. For larger data sharing, different methods should be used like: DMAing data, cache flushing etc. Ideally one would only share small control structures containing minimal sets of pointers using uncached accesses
3. DDR can only be accessed via the cache infrastructure.
4. L1 instruction cache is a read-only cache so generally, apart from invalidating it at the beginning of our application there won't be much care required for this. Even this is not necessarily required as on reset the cache is invalidated.
5. L1 data cache can be used in write-through mode where it would write its data both in L2 cache and L1 cache. It can also be configured in write-back. When in write-back mode the cache needs to be flushed completely in order to get its contents into memory.
6. L2 cache may be completely bypassed by the L1 cache accesses. This can be done two ways: either by configuring the cache in bypass mode, or by configuring the L2 cache partition assigned to a particular

SHAVE as being bypassed, in case a mix of L2 cached SHAVES/L2 uncached shaves is desired.

Considering the above description and notes we can imagine 7 important types of valid/recommended data sharing use cases inside the system:

1. Leon reading control (small) data from SHAVEs.
2. Leon working with large data from SHAVEs.
3. SHAVEs reading/writing control (small) data from Leon(s).
4. SHAVEs reading/writing large data from Leons.
5. Leon reading/writing control (small) data from other Leon.
6. SHAVE reading/writing control (small) data with other SHAVE.
7. SHAVE reading/writing large data with other SHAVE.

We'll discuss all of these separately.

4.3.2.3 Leon reading control (small) data from SHAVEs

When this is required the recommended way is to use CMX uncached view addresses. These are the ones starting with 0x78xxxxxx. This may be done in the MDK by placing such shared variables in the .cmx_direct.data section. Even if passing pointers to the SHAVE, the pointers should point to variables declared as such, or allocated in heaps staying in these addresses. Pointer arithmetic options may also be used by masking CMX variables with 0x08000000.

4.3.2.4 Leon working with large data from SHAVEs

In this case generally data would be shared via DDR. There are two ways this can be done:

4.3.2.4.1 Using DMA

Using the CMX DMA engine, data would go to and from SHAVE into DDR memory in an uncached way. Data in memory would then be uncached. However, we still need to be careful about the Leon cache infrastructure. In order to avoid any data still being stale in caches, it would be required to invalidate both the L1 cache and L2 cache.

NOTE: In this particular case, one has to make sure **L2 is configured in write-through mode.** This is because one cannot issue INVALIDATE_AND_WRITE_BACK commands because these may overwrite the data written by the SHAVE, instead, only invalidates are allowed in this case. But if so, the only way of ensuring that all the data is already in memory, is if the cache was configured in write-through mode.

4.3.2.4.2 Using DDR via L2 cache

If the SHAVE has added data in DDR via L1/L2 cache infrastructure, a series of steps need to be performed by the Leon each time before reading this data:

1. Flush and invalidate L1 SHAVE cache to L2 SHAVE cache (in case SHAVE L1 data was enabled too).
2. Flush and invalidate the SHAVE L2 data partition associated to the current SHAVE.

After the steps above are done, the procedure is similar to using DMA: L1 and L2 cache would need to be invalidated. The same note as above applied here: **L2 cache must be configured in write-through** mode for this sharing to work.

4.3.2.4.3 SHAVEs reading/writing control (small) data from Leon(s)

In this case, the recommendation is to use CMX uncached view memory.

4.3.2.4.4 SHAVEs reading/writing large data from Leons

Usually the Leon processor would launch the SHAVE before this would be required. In this case, before launching it, the Leon should make sure to invalidate the SHAVEs L2 cache partition and L1 data cache (if L1 data cache is on). Once the SHAVE has started, the SHAVE can simply read this data without other special calls.

4.3.2.4.5 Leon reading/writing control (small) data from other Leon

In this case too, sharing should be done using uncached views. If it is simply data shared between Leons and not shaves, either one of .cmx_direct.data or .ddr_direct.data sections may be used for sharing. If the data is also shared with a SHAVE, only .cmx_direct.data is recommended.

4.3.2.4.6 SHAVE reading/writing control (small) data with othe SHAVE

This should be done by using variables set in the 0x78xxxxxx address space.

4.3.2.4.7 SHAVE reading/writing large data with other SHAVE

The recommended way in this case is to use CMX DMA. Alternatively, SHAVES can share using the cache infrastructure, but in this case they would have to signal the Leon Processor to perform L1/L2 cache invalidating before different accesses happen.

4.3.3 SHAVE L2 cache partitioning

The SHAVE L2 cache supports partitioning. This means the cache may be configured from 1 to 8 partitions of sizes: 16KB, 32KB, 64KB, 128KB or 256KB.

- Any combinations are possible as long as it is a maximum of 8 partitions and maximum size is 256 KB (the L2 cache size).
- Each partition may be assigned either to the LSU (load-store) units of a SHAVE or to the instruction port of a SHAVE.
- A partition may have multiple SHAVE code/data assigned to it.

Considering the above, there are a few recommended practices about L2 cache partitioning:

4.3.3.1 Share instruction partition where code is shared

When multiple SHAVEs execute the exact same DDR code, it is a good advantage to configure all these shaves to one single L2 cache partition ideally bigger than or equal to the code being run by the SHAVEs. For example, if multiple SHAVEs share a 100 KB code, they could all be assigned to a same declared 128 KB L2 cache partition.

4.3.3.2 Share data partition where possible

If multiple SHAVEs have relative data locality with their algorithms, it is a good advantage to configure the shaves into the same data partition.

4.3.3.3 Avoid partition thrashing

If different algorithms do not have any consistent data locality, then it is recommended to just assign smaller data partitions to each SHAVE to avoid cache thrashing.

4.3.4 Debugging cache coherency issues

Generally, if one suspects cache coherency as being a problem in their application, the way to determine this for sure is by rerunning the application with all possible caches either disabled or bypassed. The most

efficient way of doing this is to run from uncached CMX view. If this is not possible for various reasons, rerunning the application in this configuration may also help the investigations:

- Leon L2 cache disabled.
- Leon L1 cache disabled.
- SHAVE L2 cache bypassed.

Running this way though also means the application would run orders of magnitude slower, this in turn could make an application work even if it is not necessarily broken because of caches. Another alternative is to ensure all chapter 4.3.1 recommendations are met, and all flush functions are called.

4.4 CMX

4.4.1 Overview

CMX is an acronym that stands for Connection Matrix Crossbar.

The CMX memory of 2 MB may be considered as 16x128 KB 'slices'.

Slice	Start Address	End Address
0	0x70000000	0x7001FFFF
1	0x70020000	0x7003FFFF
2	0x70040000	0x7005FFFF
3	0x70060000	0x7007FFFF
4	0x70080000	0x7009FFFF
5	0x700A0000	0x700BFFFF
6	0x700C0000	0x700DFFFF
7	0x700E0000	0x700FFFFF
8	0x70100000	0x7011FFFF
9	0x70120000	0x7013FFFF
10	0x70140000	0x7015FFFF
11	0x70160000	0x7017FFFF
12	0x70180000	0x7019FFFF
13	0x701A0000	0x701BFFFF
14	0x701C0000	0x701DFFFF
15	0x701E0000	0x701FFFFF

4.4.2 Notes

- Each SHAVE has higher bandwidth/lower power access to its "own" local slice
- Slice locality follows SHAVE number: Shave0 is assigned the lowest 128 Kbyte of CMX, Shave11 – Slice 11. Slices 12 to 15 are not tied to any SHAVE. They may be freely used for any other purposes.
- Slice locality is a weak concept, each SHAVE can access any other slice in CMX at the same cost, but inter-slice routing resources are finite. In addition, a slave accessing data in its own slice is more energy-efficient. As such, each SHAVE has an affinity to its local CMX slice, and this is worth keeping in mind at design time for optimal performance.

4.4.3 CMX Configuration for optimized 128/64 bit access

The CMX has a configuration word that determines how the underlying RAMs are organized to implement the memory space. There are 2 configurations supported as shown in Figure 7.

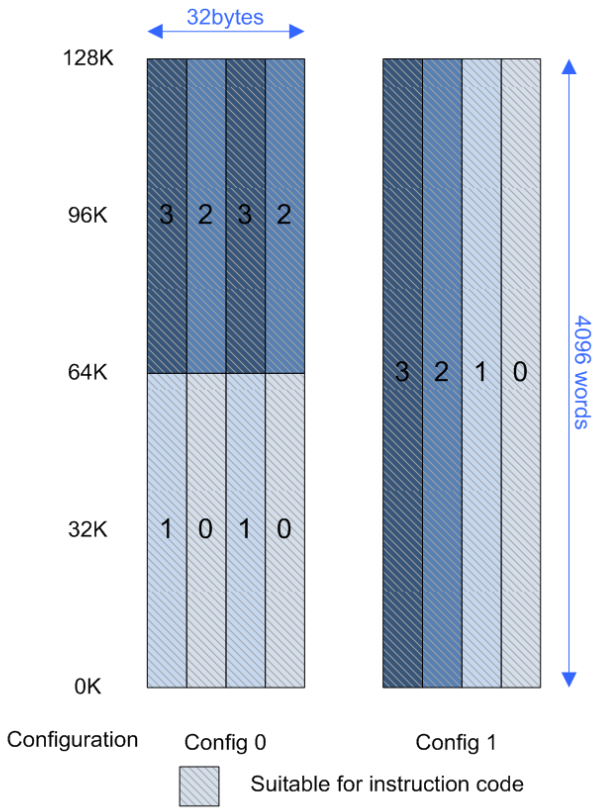


Figure 7: Supported CMX configurations

While Config 0 is useful for various HW testing, for application development Config 1 is the one which should exclusively be used. This allows for the best layout so that individual ram tiles have a minimal risk of getting accessed at the same time resulting in little to none stall cycles caused by same tile access in the same cycle.

The CMX slices may only be configured in the same configuration. It is not possible to have slice 1 in Config 0 and slice 2 in Config 1. All the CMX slices may only be configured in the same configuration.

4.5 Data Layout Optimization

4.5.1 Overview

When executing SHAVE code, stalls are occasionally observed. In most cases these stalls may be improved or resolved completely. This section details information on stall cycles that are visible when using the Myriad simulation tool.

4.5.2 Stall Sources and Mitigations

4.5.2.1 Late Read Data

Stall Source	Stall due to late read return
Why this happens	Read return data did not occur within defined time
Mitigation	• Avoid direct DDR access where possible. If required, ensure L1 and L2 caches are enabled • Access data via CMX where possible • Do not access the same CMX tile (see boxes in Section 4.4.3) from both load store ports in the same cycle • Ensure that code and frequently used data do not both reside in the same CMX tile. Try putting code into DDR and using L1 instruction cache for that. • Avoid vector accesses to unaligned addresses

4.5.2.2 BRU Miss

Stall Source	Stall due to BRU miss
Why this happens	Jump labels found at unaligned addresses
Mitigation	Align jump labels. Using the -L as moviAsm option automatically aligns jump labels at the cost of code size

4.5.2.3 Instruction fetch miss

Stall Source	IDC FIFO Low
Why this happens	If instruction bytes are not loaded fast enough
Mitigation	• If this occurs within a loop of wide hand assembled instructions, consider ways to reduce instruction width • Consider where the instructions are stored in the memory, does another process (LSU) frequently access the same CMX tile?

4.6 Bandwidth

4.6.1 DDR Bandwidth

On MA2150, the part used is DDR2 533 MHz, giving a theoretical max throughput of 4264 MB/sec. In practice between 60-80% of this is achievable.

On MA2450B, the part used is DDR3 732 MHz, giving a theoretical max throughput of 5856 MB/sec. In practice between 60-80% of this is achievable.

4.6.2 On Chip Bandwidth

The high on-chip bandwidth with 128/64-bit `AXI` and independent read/write buses means that on-chip bandwidth is rarely a limiting factor.

5 MDK build system

5.1 Introduction

This chapter provides a general overview of the build system. The build system offers the means to build an application, the means to configure it and some functional targets.

Main functionality is done already in the common makefiles (found in `mdk/common/*.mk`) but some actions are required also in the project Makefile.

5.2 Overview of the build flow

5.2.1 Example diagram of building an application

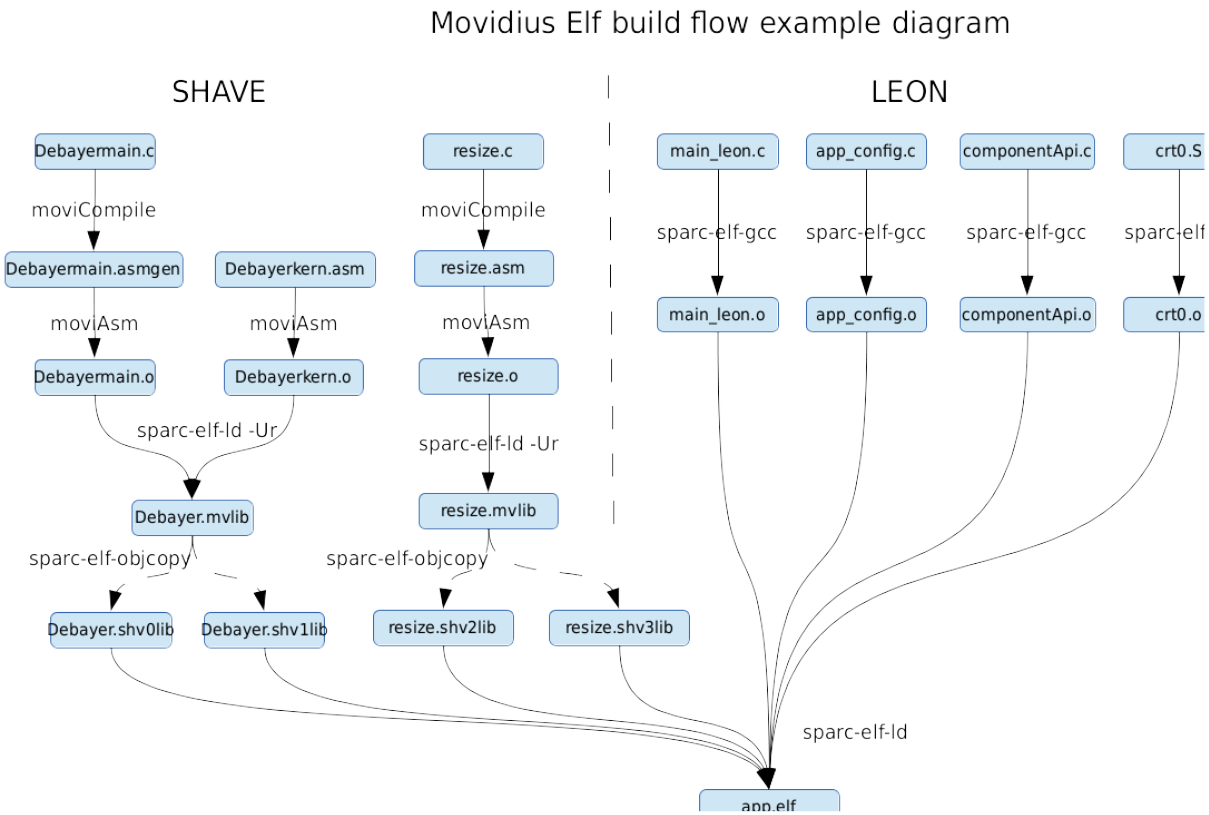


Figure 8: Application building example

5.2.2 General concepts

The ELF development flow consists of the following conceptual process steps:

1. **SHAVE**
   a. Compile and assemble the sources that will form the SHAVE component of the program.

   b. Link the resulting object files along with the required SHAVE object libraries into a single object file that is still "relocatable" – that is, it requires further linking to finalize.

2. **LeonOS**
   a. Compile and assemble the sources that will form the LeonOS component of the program.
   b. Link the resulting object files along with the required LeonOS object libraries into a single object file that is still "relocatable" – that is, it requires further linking to finalize.
3. **LeonRT**
   a. Compile and assemble the sources that will form the LeonRT component of the program.
   b. Link the resulting object files along with the required LeonRT object libraries into a single object file that is still "relocatable" – that is, it requires further linking to finalize.
4. **Final Link**
   a. Decide where each component should be deployed.
   b. Create the final fully relocated program image for deployment on the chip.

Because there are 2 x Leons (LeonOS and LeonRT), it is necessary to keep their contributions to the complete program separate from each other. To achieve this, one or both of the pre-linked LeonOS or LeonRT components needs to have its sections and symbols renamed so that they do not conflict with the names from the other Leon component.

Similarly, because there are 12 x SHAVEs, the code for each SHAVE forming the complete program needs to have its sections and symbols renamed so that they do not conflict with each other. As a side-effect of this, the decision about which SHAVE the component is to execute on is decided during this renaming; and furthermore, the programmer can decide to have two or more instances of the same component contribute to the whole program by renaming the same pre-linked source SHAVE component differently for each SHAVE.

Because there are 14 processors, and up to 14 program components, the renaming has to occur on either ALL or on all but one of the components. This is because simple naming conflicts can occur, not just between processors with the same architecture [SHAVE:SHAVE or Leon:Leon], but between processors with different architectures [SHAVE:Leon]; with common symbols having names like '`malloc`' or '`printf`'. So symbol renaming is essential to ensure that each component does not cross contaminate other components that may have conflicting names.

Section renaming on the other-hand is there to allow the programmer to "place" each of the program elements in the appropriate areas of memory; for example, placing the data for SHAVE #5 in the CMX SLICE for SHAVE #5. By employing the section renaming tactic, this permits the program to be carefully laid out in memory.

The MDK uses the convention of renaming all sections for SHAVE N by prefixing it's sections with "`.shvN.`" and similarly by prefixing its symbol names with some other value, often "`<appname>N_`" where '`<appname>`' is the name of the application.

So when the renaming has been completed, all of the components of the final program can be safely linked together to form the single program binary image for deployment on the Myriad chip.

The implementation of the build flow requires the use of a modified `sparc-elf-gcc` suite, `moviCompile` and `moviAsm`, out of which the last two are fully internally developed tools.

The `LEON` C source files (for example `main_leon.c`, `app_config.c` and `componentApi.c`) are compiled with sparc-elf-gcc resulting in sparc elf objects.

The `LEON` assembly files (for example `crt0.S`) are assembled with sparc-elf-as, resulting in sparc elf objects.

The `SHAVE` code is split in two shave applications, named debayer and resize. The `SHAVE` applications have to be build separate for each project, each project can have one or many `SHAVE` applications.

Shave C source files are compiled with `moviCompile`, internally developed tool. Next step is to make assembly the project .asm files and the .asm `moviCompile` generated files with `moviAsm`, another tool developed internally. After the assembly step all `SHAVE` chosen files will be sparc-elf object files.

5.2.3 First SHAVE link phase

Each MDK application needs to use the new SHAVE First Phase Link LD script which is provided with the compiler when performing the "pre-link" stage for aggregating the SHAVE source files.

This link phase should also specify the following options:

```
-Ur -L <path-to>/shave_first_phase.ld --gc-sections
```

plus ONE of the following additional options:

```
-e __start
```

or:

```
-u __crtinit -u __crtfini [{-u <your-Leon-called-entry-points>}*]
```

The first of these two options supports the conventional ISO C/C++ execution model where a program is executed by series of preliminary initialization steps, followed by the execution of the dynamic initializers of the static extent objects ("static extent" is related to "global", but has a few nuances that make it different), then the function '`main`' is called. This step is often referred to as the '`crtinit`' or "C RunTime INITialisation" step. The function '`main`' has two modes for terminating (exception handling is not supported by '`moviCompile`'):

- '`abort`' might be called. This means that something very bad happened, and the program should cease execution immediately. The implementation of 'abort' should report to the host that the program has terminated early with some information about why – usually a message to the console.
- '`exit`' might be called. '`return <integer-value>;`' from 'main' and '`exit(<integer-value>)`' from anywhere else are the same. In this case the programmer is informing the system that the program is complete. The '`exit`' function will execute all functions registered with '`atexit`' (C and C++) and also the destructors for all initialized objects with "static extent".

In the current implementation, both '`abort`' and '`exit`' call '`_exit`' after completing their work. So if you have a conventional program with a normal 'main', use this option when performing the first phase link.

The second option is the one of greater significance to general MDK applications. A program for the Myriad platform using the MDK has a different execution model to the conventional C/C++ execution model in that the control-flow no longer flows naturally from '`main`' continuing until the program has completed. Instead it uses a flow that is more like an "Event Driven" or "Task Driven" system. In these models a higher executive determines what functions are to be called, and in the Myriad MDK this executive is the Leon part of the program.

So from the perspective of the SHAVEs, there is no sequence of function calls and they are in effect asynchronous with respect to each other. This is where the set of '`-u`' options INSTEAD OF the single '`-e`' option is relevant.

5.2.4 Some more details about garbage collection and anchoring symbols

The use of the '`--gc-sections`' option works with the '`-ffunction-sections`' and '`-fdata-sections`' options which are used with the compiler to direct it to place each function and each data object in a uniquely named section (this is the default behaviour for '`moviCompile`'). So a single ELF object file might contain many uniquely named sections, with each section defining a single unique symbol.

The linker “resolves” all required symbols from either the set of provided ‘.o’ object files, or from one of the provided libraries. After it has resolved all of the symbols which are required, it then looks at all the sections it has collected, and for each uniquely named section in which none of the required symbols is defined, that section is deleted.

This can considerably reduce the amount of memory required by a program as the linker is able to discard all contributions it can “prove” are not actually required by the program. And this is where entry points come in.

The default linker script (LD script) is pre-built into the linker, and this usually has a single “Start” symbol. In fact, for most Linux systems this is a symbol named either ‘_start’ or ‘__start’, and it is the name of the very first instruction to be executed by the program in the ‘crtinit’ code. The ‘-e’ option is a way for the programmer to explicitly provide an alternative name for the start symbol.

The linker is then required to find and keep this symbol definition, and it must also keep all symbols referenced by that definition, and transitively for all other symbols referenced as a consequence of this.

If the linker has no start symbol, it has no reason to keep any symbols at all, and the garbage collector will happily delete the whole program!

The model of a single start symbol works perfectly well for the conventional C or C++ program, but for a Myriad program using the asynchronous execution model “each” entry-point is effectively a start symbol, none of which might otherwise be referenced in a way that will prevent the garbage collector from eliminating them.

This is where the set of ‘-u’ options is required. Each entry-point must be identified to the linker using a ‘-u’ option so that it knows that it must keep that symbol, and thus all other symbols directly or indirectly referenced through these symbols. They are in a sense the “anchors” for the linker, which it then uses to determine what should and should not be kept during garbage collection.

It is of course possible to disable the garbage collector and avoid the pain of adding these entry-point names as options to the linker, but this could see the program use far more memory than is necessary, and in many MDK examples this could be as much as a 40% penalty, which is a large amount of precious CMX memory to waste.

The default shave_first_phase.ld linker script will aggregate all of the sections with the following names:

```
[.gnu.linkonce][{.cmx|.ddr}]{.text|.data|.rodata|.bss}[.*]
```

Producing output sections with the following names:

```
S[{.cmx|.ddr}]{.text|.data|.rodata|.bss}
```

So the final link-phase need only deal with these specific names, plus any user-defined section names, all grouped into aggregated sections which are then better managed by later stages of linking.

5.2.5 Final steps

Next step is to map the SHAVE application(s) onto the cores needed. In the example from Figure 8 this is done by making shvXlib, shvXuniqlib, shvXwndlib and shvdlibshvdcomplete. In this example the SHAVE applications are shvXlib files. If we summarize all of the above sections, when getting to this step we have the shave code with aggregated sections and all that is required is to perform the section and symbol renaming discussed in section 5.2.2.

The final step is to link all sparc-elf objects created, LEON and SHAVE ones, into app.elf executable using sparc-elf-ld. At this step if no local *.ldscripts are chosen the default ones will be used, which may be found in mdk/common/scripts/ld.

5.3 Main build targets

5.3.1 Introduction

The MDK build system contains a suite of available targets in order to build the final executable elf file. These common Makefile includes may be found in `mdk/common/*.mk` .

5.3.2 Available targets

Targets are split into different files according to their role.

1. `generic.mk`

   This is the makefile that should be included by the user's makefile. Includes all necessary makefiles, files that are listed and detailed below.

   Targets available:

   - all – the start target, has dependency on the application final elf and mvcmd file, also on the `LEON` and `SHAVE` listing files.
   - $(DirAppOutput) / $(APPNAME) .map – target dependent on the $(DirAppOutput) / $(APPNAME) .elf file
   - %.o – target to make all `LEON` object files from corresponding %.c/%.S using sparc-elf-gcc
   - % `_shave.o`
     - targets to make `SHAVE` sparc-elf objects from corresponding %. `asmgen` , these rules use `moviAsm` . One target is for `SHAVE` sparc-elf objects without debug information, and one with debug information in the case that the user defines SHAVEDEBUG = yes
     - target to make sparc-elf objects from corresponding %. `asm` file using `moviAsm`
   - %. `asmgen` – targets that generate `SHAVE` assembly files from corresponding %.c/%. `cpp` using `moviCompile`
   - %. `i` – target that generates a pre-processed C file from corresponding %.c
   - %. `ipp` – target that generates a pre-processed CPP file from corresponding %. `cpp`

   NOTE: More information about preprocessed files and their valuable purpose may be found in the moviCompile manual.

   - %.shv0lib, %.shv1lib, %.shv3lib, %.shv4lib, %.shv5lib, %.shv6lib,%.shv7lib, %.shv8lib, %.shv9lib, %.shv10lib, %.shv11lib – each target corresponds to a `SHAVE` and maps the corresponding %.mvlib to a shave using sparc-elf-objcopy with options to prefix symbols (with `nameOfTheApp` + `correspondingShaveNumber` +_) and options to prefix sections(with ".shv"+ `CorrespondingShaveNumber`)
   - %.shv0wndlib, %.shv1wndlib, %.shv3wndlib, %.shv4wndlib, %.shv5wndlib, %.shv6wndlib, %.shv7wndlib, %.shv8wndlib, %.shv9wndlib, %.shv10wndlib, %.shv11wndlib - `each target corresponds to a shave and maps the corresponding %.mvlib to a shave using sparc-elf-objcopy with options to prefix symbols (with nameOfTheApp+correspondingShaveNumber+_) and options to prefix sections (with ".wndshv"+CorrespondingShaveNumber)`
   - %.shvdlib – target to create a dynamically loadable library from the corresponding %.mvlib using sparc-elf-ld
   - %.shvdcomplete – target to create a windowed library for symbol extraction using the corresponding %.mvlib using sparc-elf-objcopy to create symbols file for the dynamic section.
   - % `_sym.o` – target that uses %. `shvdcomplete` file with spa

- % _raw.o – target to make a raw object from %. shvdlib using sparc-elf-objcopy, this rule puts the data into . ddr_direct.data section
- $(DirAppOutput) /%.mvcmd – target to make the mvcmd boot image using moviConvert
- $(LinkerScript) - target that generates final linker script
- $(LEON_RT_LIB_NAME) .mvlib – target with dependency on $(LEON_RT_APP_OBJS) $(LEON_SHARED_OBJECTS_REQUIRED) $(ALL_SHAVE_APPS) which uses sparc-elf-ld
- %.rtlib – target that generates a unique instance of LEON RT application using sparc-elf-objcopy from corresponding %.mvlib file
- localclean – target to delete project specific built files clean – target to delete all built files from the MDK distribution and all project built files.

2. generalsettings.mk

   Contains definitions of variables that hold paths to common and application folders.

3. toolssettings.mk

   In this file can be found all variable definitions needed for shave and LEON tools.

4. includessettings.mk

   In this file are the variables that keep the include definitions for LEON and SHAVE code.

5. commonsources.mk

   Contains definitions of variables that build:

   - list of paths to LEON and SHAVE component folders
   - list of paths to LEON and SHAVE include component folders
   - list the application specific leonOS C and asm sources
   - list the application specific leonRT C and asm sources
   - list the application specific leonRT objects
   - list of LEON component C and asm sources
   - list of LEON driver C and asm sources
   - list of SHAVE applications, for each SHAVE
   - list of LEON component headers
   - list of LEON headers
   - list of SHAVE component headers
   - list of SHAVE headers
   - list of SHAVE asm headers, *. incl files
   - list of LEON shared object files
   - list of LEON application object files
   - list of all SHAVE applications

6. commonbuilds.mk

   In this file can be found the build rules for the SHAVE commonly used libraries. Also contains definitions of variables that build list of SHAVE C, CPP and asm objects from SHAVE code that can be found in the common folders and definitions of variables that build list of SHAVE C, CPP and asm objects from components. In the second part of this file are targets used to make listing files.

7. mbinflow.mk

   Contains targets to make *. mobj files from corresponding *. asm /*. asmgen files. A mobj file or Movidius Object File is a binary file containing all the sections of the source codes, it is obtained with moviAsm using moviAsm options defined in includessettings.mk and in toolssettings.mk .

The `mbinflow.mk` has also a target for * `_raw.o` , this type of objects are obtained from corresponding *. `mbin` using sparc-elf-objcopy.

8. `functional_targets.mk`

   The targets of this file will be detailed in section 5.7.

5.3.3 Mvlibs

The mvlibs may be built two ways: automatic or manual. The automatic method varies depending on static or dynamic data loading methods. For the dynamic data loading method of automatically having mvlibs built, please refer to the dynamic data loading description.

In the case of static data mvlibs in order to automatically have the build system trap and build your mvlibs it is sufficient to place them in separate folders inside your project's shaveApps folder. For example, the following structure:

```
./
|-- Makefile
|-- Readme.txt
|-- config
|   |-- custom.ldscript
|   `-- kernel_test.ldscript
|-- leon
|   |-- app_config.c
|   |-- app_config.h
|   |-- main.c
|   |-- measureTask.c
|   `-- rtemsConfig.h
|-- leon_rt
|   |-- app_config.c
|   |-- app_config.h
|   `-- main.c
|-- shared
|   `-- common_includes.h
|-- shave
|   |-- shave.c
|   `-- shave.h
`-- shaveApps
    |-- absoluteDiff
    |   `-- shaveMain.c
    |-- accumulateFp16
    |   `-- shaveMain.c
    |-- arithmeticAdd
    |   `-- shaveMain.c
```

would automatically define mvlibs for three applications: absoluteDiff.mvlib, accumulateFp16.mvlib and arithmeticAdd.mvlib. Automatic rules require at least one entry point named “Entry” to be defined. It is also possible to have manual building rules. The following sections describe manual rules in more details. In the case of manual rules the build rules for the required mvlibs are filled in by each individual project. These rules will have to be defined in the project local Makefile.

5.4 Makefile: LEON code

5.4.1 Suggested starting approach

The recommended approach to starting a new application is to copy the `mdk/examples/myriad2/Progressive/000_HelloWorld_BareMetal` application and then trim out any unneeded code. However, if starting from scratch, the application's Makefile needs to conform to some rules in order to make it conformant to the rest of MDK projects. This helps significantly in requesting support.

5.4.2 Including the MDK build flow functionality prerequisites

Each application Makefile in the MDK has to include the following:

COMMON definitions:

```
# Set MV_COMMON_BASE relative to mdk directory location (but allow
user to override in environment)
MV_COMMON_BASE  ? = ../../common

# Include the generic Makefile
include $(MV_COMMON_BASE) /generic.mk
```

This provides access to the MDK common build flow settings.

5.4.3 Tweaking build options

If any special macros need to be added these may be added directly in the Makefile at the end of the file. For example:

```
# ------------------------- [ Build Options ] ---------------------- #
# App related build options
# Extra app related options
CCOPT          + = -DDEBUG
MVCCPPOPT      + = -DNDEBUG
CCOPT_LRT      + = -U'DEFAULT_HEAP_SIZE'
MVCCOPT        + = -D'SIPP_USE_MVCV'
MVCCOPT_LRT    + = -D'SIPP_USE_MVCV'
```

5.4.4 Including components

Components may be included by simply enumerating them into the `ComponentList` variable. For example:

```
# ------------------------[ Components used ]-----------------------#
ComponentList = MvCV MvSTL CifGeneric MV0212 LcdGeneric HdmiTxIte
CifOV5642
```

In case any components have shave code that needs to be added into a library before linking, this can be done by specifying:

```
#----------------[ Local shave applications sources ]-------------#
#Choosing if this project has shave components or not
SHAVE_COMPONENTS =yes
```

5.5 Makefile: SHAVE code

5.5.1 Adding SHAVE code

To start adding `SHAVE` code, one must first decide what apps are required. Once the apps are defined, for each app, a set of rules may be copied over from one of the existing examples, for example from `ShaveHelloWorld`:

```
#-----------------[ Local shave applications sources ]--------------#

#Choosing C sources the hello application on shave
HelloApp = shave/hello
SHAVE_C_SOURCES_hello = $(wildcard $(DirAppRoot) /shave/*.c )
#Choosing ASM sources for the shave hello app on shave
SHAVE_ASM_SOURCES_hello = $(wildcard $(DirAppRoot) /shave/*.asm )

#Generating list of required generated assembly files for the hello
app on shave
SHAVE_GENASMS_hello = $(patsubst %.c,%.asmgen, $
(SHAVE_C_SOURCES_hello) )
#Generating required objects list from sources
SHAVE_hello_OBJS = $(patsubst %.asm,%_shave.o, $
(SHAVE_ASM_SOURCES_hello) ) \
                   $(patsubst %.asmgen,%_shave.o, $
(SHAVE_GENASMS_hello) )

#update clean rules with our generated files
PROJECTCLEAN + = $(SHAVE_GENASMS_hello) $(SHAVE_hello_OBJS)
#Uncomment below to reject generated shave as intermediary files
(consider them secondary)
PROJECTINTERM + = $(SHAVE_GENASMS_hello)
```

The “hello” and “HelloApp” strings need to be changed accordingly. Also, the `SHAVE_C_SOURCECS_` [name] and `SHAVE_ASM_SOURCES_` [name] variables don’t necessarily have to be defined using wildcards. They may also be defined as a simple file list.

In order to avoid having to write paths, one can use rules to select source files in a simpler way. An example of this is the `sipp/VideoEffects` example:

```
#C files that should make up the sipp filter
SHAVE_C_Filter = svuNegative.c sippShave.c svuSobel.c
#ASM files that should make up the sipp filter
SHAVE_ASM_Filter = Convolution3x3.asm Sobel.asm

#for each selected file pull in the file with path. The $(sort ...)
part is to eliminate duplicates
SHAVE_C_SOURCES_videoeff  = $(sort $(foreach file, $(SHAVE_C_Filter) ,
$(wildcard $(patsubst %,%/ $(file) , $(SIPP_PATHS) ))))
#Choosing ASM sources for the shave hello app on shave
SHAVE_ASM_SOURCES_videoeff  = $(foreach file, $(SHAVE_ASM_Filter) , $
(wildcard $(patsubst %,%/ $(file) , $(SIPP_PATHS) )))
```

Here, the user has listed the files needed for filter used and used rules that may be blindly copied anywhere to generate a simple list of files.

5.5.2 SHAVE application build rules

- Apps that don't use compiler libraries or SHAVE components

These should have a rule like:

```
#---------------[ Local shave applications build rules ]------------#
#Describe the rule for building the HelloApp application. Simple rule
specifying
#which objects build up the said application. The application will be
built into a library
$(HelloApp) .mvlib : $(SHAVE_hello_OBJS)
    $(ECHO) $(LD) -r $(SHAVE_hello_OBJS) -o $@
With "HelloApp" and "hello" changed accordingly.
```

- Apps that use compiler libraries or SHAVE components

```
-------------------------------[ Local shave applications build
rules ]------------------#
#Describe the rule for building the ImageTwoKernApp application.
Simple rule specifying
#which objects build up the said application. The application will be
built into a library
$(ImageTwoKernApp) .mvlib : $(SHAVE_imagetwokern_OBJS) $(PROJECT_SHAVE_LIBS)
   $(ECHO) $(LD) -r $(SHAVE_imagetwokern_OBJS) $(PROJECT_SHAVE_LIBS) -o $@
```

5.6 Makefiles: Linking SHAVE apps to cores

Once apps packaged in *. mvlib files exist, these can be placed onto cores resulting SHAVE libraries (for example shXlib , shvXuniqlib).

It is necessary to enumerate the *.mvlib 's into SHAVE_APP_LIBS variable.

```
#-----------------[ Shave applications section ]----------------#
SHAVE_APP_LIBS = $(ImageTwoKernApp) .mvlib
```

Rules necessary to build the SHAVE libraries are in the common makefiles, user just has to define SHAVEX_APPS needed.

Each SHAVE application can be mapped on any SHAVE

```
SHAVE0_APPS = $(resize) .shv0lib
```

or on many SHAVE s:

```
SHAVE2_APPS = $(resize) .shv2lib
SHAVE3_APPS = $(resize) .shv3lib
```

One SHAVE can have more than one SHAVE application.

```
SHAVE0_APPS = $(resize) .shv0lib $(Debayer) .shv0lib
```

5.6.1 Using shvXlib

To place apps on cores using absolute addresses and individual symbols, one needs to add the apps for each SHAVE using *.shvXlib names. For example:

```
#-----------------[ Shave applications section ]-----------------#
```

```
SHAVE_APP_LIBS = $(ImageTwoKernApp) .mvlib
SHAVE0_APPS = $(ImageTwoKernApp) .shv0lib
```

5.6.2 Using shvXuniqlib

These libraries should be used when in need of merging symbols into one and sharing them. When using `shvXuniqlib` files, an extra rule needs to be placed in the file:

```
#describe the unique libraries rules. For this case only the shared1
and shared2 symbols need
#uniquifying
%.shv2uniqlib : %.shv2lib
    $(ECHO) $(OBJCOPY) --redefine- syms = $(SYMBOLS_UNIQ_SHAVE2) $< $@
And the apps variables also need to be selected as before. For
example:
#----------------[ Shave applications section ]--------------#
SHAVE_APP_LIBS = $(global_sharing) .mvlib
#"uniq" libraries are libraries the user defines for himself which
keep
#some symbols unique so that they may be shared between many apps
SHAVE2_APPS = $(global_sharing) .shv2uniqlib
SHAVE3_APPS = $(global_sharing) .shv3uniqlib
```

A clean way of writing rules and selecting sources to keep files to a minimum length is demonstrated in the `mdk/examples/myriad2/HowTo/SharingExample` MDK example using .mk files.

5.7 Build system functional targets

Apart for its role as a build system and providing configuration options, the MDK Makefile structure allows also for a few functional targets to ease development. These are listed below.

5.7.1 make help

Displays a help message containing a short description of all `make` targets.

5.7.2 make all

Used to build a target after changes were done to C or assembly source files.

WARNING: `ldscript` files and `Makefiles` are not covered by “ `make all`”. If changes are done to the `ldscript` files or `Makefiles`, the build needs to be cleaned first before running “`make all`”.

5.7.3 make run

Build everything and run it on a target via `moviDebug`. `MoviDebugServer` needs to be started previously for this target to work correctly. Alternatively, `moviSim` may also be started instead of `moviDebugServer` if the user wishes to run on the simulator using `moviDebug`.

5.7.4 make start_server

Starts `moviDebugServer` required to run `moviDebug`. This may alternatively be called with: “`make srv`”.

5.7.5 make start_simulator

Starts `moviSim` in quiet mode for testing with the simulator.

5.7.6 make start_simulator_full

Starts `moviSim` in verbose mode (full log). Recommended usage: make `start_simulator_full > results.log` . Traps all verbose log in "`results.log`" file.

5.7.7 make debugi

Starts `moviDebug` for interactive use.

5.7.8 make debug

This debug target differs from the run target in that it strips out any call to exit in the run script. This allows the user to do interactive debugging once the execution has stopped or alternatively to hit CTRC+C during the `runw` session and to start checking the state of execution.

5.7.9 make report

Creates a graphical report of memory utilization. The output is an html file which can be opened in any browser. Application has to be built previously.

5.7.10 make clean

Clean build files.

5.7.11 make load

Builds application and loads target elf into debugger but doesn't start execution (allows setting breakpoints in advance).

5.7.12 make flash

Program SPI flash of the MV0xxx board with your current `mvcmd`.

Points to note:

- Do not have any printfs in the application to be flashed. Building using 'make clean all `NO_PRINT`=YES' ensures this is the case. Alternatively, `-DNO_PRINT` as a C option in the Makefile can be used.
- Flashing erases the entire flash, and then program minimum number of blocks required to write the desired image. The image is then read back and validated.

5.7.13 make flash_erase

Erase SPI flash of the MV0xxx board.

Please note:

- 'make flash_erase' erases the flash. Recommended prior to interactive debugging.

5.7.14 make show_tools

Displays tools paths.

5.8 Memory mapping management

5.8.1 Overview

In many cases memory adjustments need to be made to fit code or data at the correct memory address or to remap them towards a desired memory storage device. This can be done by using custom linker scripts, that allow to link symbols to a desired memory address.

In case of the MDK this can be done by creating a folder named “config” in the project root folder, and copying in this folder one of the standard linker scripts dependent on the socket variant the project is built for.

5.8.2 Standard linker scripts provided in MDK

There are 3 standard linker scripts provided:

- “`myriad2_default_memory_map_elf.ldscript`”, by default includes the linker script defined for socket revision ma215x;
- “`myriad2_default_memory_map_ma215x.ldscript`“, configured for DDR_SIZE 128 MB;
- “`myriad2_default_memory_map_ma245x.ldscript`“, configured for DDR_SIZE 512 MB.

These files can be found in the MDK release bundle under the following path:

```
.common/scripts/ld/myriad2memorySections/
```

5.8.3 Recommend usage of the linker scripts

It is recommended to use the files defined in section 5.8.2 as a template for custom memory linking script. First copy the file “`myriad2_default_memory_map_elf.ldscript`” in the created “`config`” folder then rename it to: “`custom.ldscript`”. Then copy[1] one of the appropriate “`myriad2_default_memory_map_*.ldscript`“ files that ends with the name of the socket revision of the project and make sure this file name is included by the “`custom.ldscript`” file that was first copied. The MDK build system will automatically identify and use this “`custom.ldscript`” linker configuration script file at build time.

The “`custom.ldscript`” includes all the necessary `ldscript` files that contain the defined memory sections and the rules for each of them that specifies what input and output sections are linked in them.

The socket revision specific ldscript file structure is very simple. It starts with the definition of the MEMORY sections, as a default MDK providing the following MEMORY sections defined:

- For each shave there is a code and separate data section named according to the following rule: SHVx_CODE, SHVx_DATA, where x represent the number of the shave from 0 to 11 in case of MA2x5x socket revision.
- For Leon OS code and data in one section named LOS.
- For Leon RT code and data in one section named LRT.
- For CMX DMA descriptors the section CMX_DMA_DESCRIPTORS.

1 **NOTE:** It is possible to remove the include command of the socket revision specific ldscript and instead copy paste its content in to the “custom.ldscript” before the other include commands.

- For other CMX related variables, code the CMX_OTHER is provided.
- For data that needs to be stored in DDR there is the DDR_DATA section provided.

These memory sections can reside anywhere in memory, as long as they don't overlap another section. Each section has a property assigned to it (wx) for sections that can be written and where executable code can live, (w) for writable sections (no executable code). Each section must have an ORIGIN = address where the memory section starts from, and LENGTH = specifying the length of the memory section.

In the properties ORIGIN and LENGTH simple mathematical expressions using + or – can be used, values are automatically recalculated if they are appended with K, M. In this case a numerical value of 1K = 1024 bytes, 1M = 1024K = 1048576 bytes.

It is possible to use hexadecimal format by starting the number with 0x.

The general rule is that shave code executes faster from CMX slice corresponding to that shave because the next instruction is loaded faster, the same principle applies for shave data, it is loaded the fastest if the data is in the cmx slice corresponding to the specific shave (because these slices have there own communication buss with the corresponding shave, read more in section 4).

Because of the faster CMX memory the same principle applies to LOS and LRT sections. Data access and code execution is faster in CMX then it would be if the code /data would be mapped in DDR.

Because any section can be remapped theoretically it's possible to force and application that has data assigned to section ddr_data to cmx by setting a CMX address + corresponding length to the DDR_DATA memory section.

The general syntax of the linker scripts is GNU LD format, if interested one can find more on this syntax by searching online and in the official GNU LD documentation at the following web address:

https://sourceware.org/binutils/docs-2.26/ld/index.html

5.8.4 Moving code or data to DDR

In order to move any memory section, for ex LOS, LRT or any Shave code or data to DDR it is enough to change the starting address to an address that is inside the DDR (see 4.2) for example in order to move LOS section to ddr one might want to change the ORIGIN address of memory section LOS to 0x8xxxxxx and then adjust the DDR_DATA section ORIGIN and LENGTH to not overlap the LOS:

```
DDR_DATA   (wx) : ORIGIN = 0x80000000 + 128K * 3, LENGTH = 128M – 128K
* 3
LOS (wx) : ORIGIN = 0x80000000 + 0*128K, LENGTH = 128K * 3
```

The same way it is possible to move Shave code or data by adjusting the ORIGIN of the `SHVx_CODE` or `SHVx_DATA`.

5.8.5 Inserting custom sections

It is possible to define additional memory sections the same way the standard sections had been defined, but in addition also specific rules need to be defined for the linker to know what input/output sections are to be mapped in the new memory section.

For example, if a special section needs to be added in order to obtain specific start address inside the code this can be done by inserting the following lines after the MEMORY {…} sections definitions:

```
/* the linker script code to define custom sections to be inserted in
the LOS memory section */
```

```
SECTIONS {
  .myfunc1 : ALIGN(16) {
    MYFUNC1_START = .;
    KEEP(*(.cmx.text.myfunc1))
    MYFUNC1_STOP = .;
  } > LOS

  .myfunc2 : ALIGN(16) {
    MYFUNC2_START = .;
    KEEP(*(.cmx.text.myfunc2))
    MYFUNC2_STOP = .;
  } > LOS
}
```

In the code this is used like this:

```
/* Declaration of the external variables who's addresses will be
assigned by the linker, in order to be able to read them in the code.
*/
extern u32 MYFUNC1_START, MYFUNC1_STOP;
extern u32 MYFUNC2_START, MYFUNC2_STOP;

// Declaration of the function prototypes mapped to cmx.text.myfunc<x>
#define MTRR_SUPPORT_MYFUNC_1(x)
__attribute__((section(".cmx.text.myfunc1"))) x
#define MTRR_SUPPORT_MYFUNC_2(x)
__attribute__((section(".cmx.text.myfunc2"))) x

void MTRR_SUPPORT_MYFUNC_1(myfunc1)();
void MTRR_SUPPORT_MYFUNC_2(myfunc2)();

void POSIX_Init ( void *args )
{
…
    /* bellow we can see how the addresses are used to calculate code
size. */
    u32 funcsize1 = (&MYFUNC1_STOP -  &MYFUNC1_START) * 4;
    u32 funcsize2 = (&MYFUNC2_STOP -  &MYFUNC2_START) * 4;
…
}
```

NOTE: The above code can be found in: `./examples/HowTo/mtrr_rtems/`

Notice the > LOS ending in the linker script code for the `.myfunc1` and `.myfunc2` output sections. This means that these sections will be inserted in to the memory section LOS, and because these are the first rules for this memory section these will be inserted at the beginning of the LOS memory section.

The same way it is possible to insert new memory sections and with corresponding input and output sections.

The new memory section must be defined in the same MEMORY {..} declaration section, because only one of these special zones is allowed to be used in one ldscript file.

The rule is that a memory section always must be defined before usage.

5.9 Multiple Myriad versions support in MDK

5.9.1 Overview

MDK build and directory system support s the following Myriad 2 silicon versions:

- MA2150
- MA2450

The motivation is to have a solution which allows to ship one MDK package which supports all of the Myriad 2 versions and allows to organize the drivers and components to be cross compilable between the Myriad 2 versions.

The concept of the solution is to structure the sources into different directories based on if they are version-independent common silicon or if they are applicable just to a particular version as outlined below:

- include – public common header files for all Myriad 2 versions.
- src – common source files for all Myriad 2 versions and internal include files.
- arch – folder containing the different Myriad 2 version specific files.
  - MA2150
    - include – public header files for MA2150 (could be empty if there is no MA2150 specific public API extending the common one).
    - src – sources and internal headers for MA2150.
  - MA2450
    - include – public header files for MA2450 (could be empty if there is no MA2450 specific public API extending the common one).
    - src – sources and internal headers for MA2450.

The build system automatically includes all common files and the Myriad 2 version specific files into the build.

5.9.2 Makefile variables

There is a makefile variable called MV_SOC_REV, which defines which Myriad 2 version you build for. The value of this variable is the **lowercase name of the Myriad 2 silicon version**, e.g. ma2150, ma2450.

The MV_SOC_PLATFORM makefile variable refers to the Myriad 2 platform family.

There is a possibility to run an application written for the MV0212 development board with an MA2450 Myriad 2 silicon in a compatibility mode that will emulate performances similar to that of an MA2150 for evaluation purposes. To enable this use the makefile variable MA2150_COMPAT=yes.

5.9.3 Predefined macro

The build system automatically predefines a preprocessor macro with **the capitalized string of Myriad 2 silicon version** defined by the MV_SOC_REV variable, e.g. MA2150 or MA2450.

6 Building Secure Boot Applications

6.1 Introduction

The MA2X55[2] variants of the Myriad 2 processor support secure boot operation. The secure boot mechanism uses a combination of 128 bit AES CBC encryption with ED25519 digital signature verification.

The AES encryption ensures that attackers are unable to reverse engineer the application code even if they have access to the secure boot image. The ED25519 digital signature system ensures that only boot images which have been signed with a matching ECC private key are bootable on secure chips. This ensures that the boot image hasn't been modified. In addition the code signing feature allows for a secure way to implement product streaming features. Many products use a common hardware platform and differentiate features based on firmware version. This type of system may be vulnerable to attackers upgrading the low cost product by simply updating the firmware image to use the image from a higher cost variant. By using different signature options for the two product streams, this type of attack is no longer possible.

The Myriad 2 chip contains 1024 bits of One Time Programmable (OTP) memory. This storage area is implemented using an array of fuses which can be electrically 'blown' and is also known as the “eFUSE Array”. It is used to store boot configuration and the encryption keys necessary for secure boot feature. In addition 384 bits of the eFUSE array are reserved for customer use. These bits may be used in an application specific way by a product. For example some of the bits may be used to sort a unique serial number for each device.

MA2X55 variant devices will be delivered to the customer in an “INSECURE” state. In this mode of operation it is possible to boot images which have not been signed or encrypted, however the JTAG port of the parts is still disabled. The process by which customers program the eFuses for their product involves booting a custom (insecure) image which in turn programs the eFUSE array with the boot configuration and encryption keys. Once the eFUSE has been programmed to enable security any subsequent reset will cause the part to reboot in secure mode. In this mode it will only be possible to boot images which have been both encrypted and signed using the corresponding AES and ECC private keys.

This chapter describes how the keys are generated within the MDK and also how secure boot images are generated. For further details on the internal operation of the boot process please see the "Myriad 2 MA2x5x Databook" chapters 2.2 (Boot Operation) and 2.3 (Secure boot) which provide full detail on the mechanisms employed and the Movidius Application Note on “Efuse Programming” which describes in detail how the eFUSE array must be programmed.

6.2 How to use secure boot

The Myriad 2 secure boot implementation is designed to make it trivial to move from generating a normal boot image to generating a secure boot image. The complexities of boot image generation are handled by the moviConvert utility which is able to generate a secure MVCMD from an input elf file and the entire process is handled by two straightforward MDK build targets.

By design, there is no need for the application developer to modify their application in order to make use of the secure boot feature, however some consideration is needed to accommodate the temporary DDR space needed for the decryption process. See section **6.2.2** for further details on this.

2 The digit 5 at the end of the part number indicate the secure boot option.

6.2.1 Development flow for Secure Applications

6.2.1.1 Application is developed in the same manner as any non-secure MDK application

- Development should be carried out using MA2x50 non-secure parts as these parts have JTAG enabled and thus can be debugged. In fact the only difference between MA2150-B or MA2450 and MA2x55 parts is that JTAG is disabled by default in hardware on MA2x55. This is necessary as it is not possible to have a secure implementation with JTAG enabled. Both parts contain the same ROM image and as such, even the encrypted images can be tested on MA2150-B or MA2450 parts.
- As a general rule applications which are intended for use with secure boot should adhere to software best practices for security. In particular products should ensure that there are no “remote code execution” vulnerabilities that could potentially allow an attacker to run code on the Myriad.
- When the application is ready for release to production, proceed to the following steps to generate keys and generate a secure application.

6.2.1.2 Generate Encryption Keys

To generate encryption keys for a given product the "make gen_keys" command should be executed in the application directory as follows:

```
$ make gen_keys
Generated keys:
 Public key : ./keys/ts_leon_simple.pubkey
 Private key: ./keys/ts_leon_simple.privkey
 AES key    : ./keys/ts_leon_simple.aeskey
 Private key must not be released
```

There are three files generated as follows:

File	Details
.pubkey	This is the ED25519 public key that is used by the MA2X55 to verify the authenticity of the secure boot image. This key must be programmed into the eFuses of all parts which will boot the secure image.
.privkey	This is the ED25519 private key that is used by the MDK build system to sign secure boot images. Only images which have been signed with this key will be bootable on secure parts containing the associated pubkey. It is very important that this key is stored securely and it shouldn't simply be checked into your revision control system. Typically this key will be stored on a secure workstation that is only used for the generation of final secure boot images. Furthermore it is crucially important that this key is safely backed up as without this key it will no longer be possible to generate new firmware images for products in the field.
.aeskey	This is the symmetric encryption key that is used to encrypt/decrypt secure images to prevent attacker access to the application code. As AES is a symmetric cipher this key is both programmed in the eFuses and also used by the build system to generate secure images. It should be treated the same as the privkey in terms of keeping it secure and making sure its backed up.

Table 1: Encryption Keys Generated by “make gen_keys”

6.2.1.3 Configure Production Setup for eFUSE programming

In order to build a product that uses the Movidius Secure boot technology, it is necessary to develop a custom eFUSE programming solution for your production assembly line. This involves taking the eFUSE programming example from the MDK and modifying it for your custom product needs. How this happens is quite application dependent. In the simplest case where all products have the same key the eFUSE programmer app can simply be modified to select the chosen boot configuration and encryption keys. This then creates a bootable application that programs the eFUSE when booted.

The production flow would involve having the MA2x55 boot once from a programming PC (e.g. via USB boot, or using something like a Total Phase Aardvark for SPI boot) to program the eFuses. The programming jig would need to be able to override the default state of the “Wakeup” pin to pull it low thereby selecting GPIO configured boot mode. The GPIO configured boot mode would then allow the eFUSE programming image to be loaded and blow the fuses to enable secure mode. Any subsequent boots would the be secure boots.

In the provided eFUSE programmer example the encryption key values are specified using an array. However the keys generated by the “make gen_keys” rule are raw binary files. It is possible to easily convert these binary keys to a text based header which can be included in the source of the programmer application using the Linux hexdump utility.

e.g.

```
hexdump -v -e '15/1 "0x%02X, " 1/1 " 0x%02X,\n"' application.aeskey > application.aeskey.h
```

Please see the Movidius Application Note on Efuse Programming for full details on how eFuses can be programed for a production environment.It is important to realize that eFUSE programing has some very specific and critical electrical requirements and as such it is important that the hardware solution developed for the programming process carefully adheres to the requirements specified in the Efuse Programming Application Note. It is also worth mentioning that the MV0182 and MV0212 MDK Evaluation platforms do not support eFUSE programming as the EFUSE_VDDQ rail is not supplied, as such it is not possible to experiment with Efuse programming using your evaluation board.

6.2.1.4 Building a secure application

Once the final production elf has been produced and tested the process of building a secure image is very straightforward.

To generate a final secure image for production you can use the make target 'secure':

```
% make secure
Generating Secure MVCMD boot image: output/production_app_secure.mvcmd
 From input elf file              : output/production_app.elf
 Using AES Key                    : ./keys/production_app.aeskey
 And ECC Private Key              : ./keys/production_app.privkey
```

NOTE: This build rule invokes moviConvert to process the application elf file and generate a secure boot image. The application elf file is a dependency of this build rule, so if it isn't already present it will be built.

The 'secure' and 'gen_keys' build rules can be found in:

```
mdk/common/functional_targets.mk
```

6.2.2 Memory Map Requirements for Secure Boot

While in general the conversion of an application from normal boot to secure boot is just as simple as invoking 'make secure' there is one exception which users much watch out for.

The nature of the secure boot algorithm is such that it requires the use of a temporary buffer space at the top of DRAM. This region is used to temporarily store the boot image while the image signature is verified.

As such it is not advisable to locate loadable memory sections in the top of DRAM as they will clash with this temporary buffer region. *Note:* This restriction only applies to sections which are loaded at boot time. There is no problem with regions of memory dynamically allocated by the application once it has been started (e.g. Framebuffers etc.) as once the application has started the temporary space is no longer needed.

MoviConvert automatically proportions this temporary buffer and follows the following decision flow:

- Calculate size of final SECURE MVCMD image:
  - Reserve a section of memory of this size at the top of DRAM.
- Check if any elf 'loadable' sections clash with this reserved region:
  - If so, reduce the reserved area to avoid clash.
    - If not possible (e.g. buffer at top of DDR) moviConvert generates an error.
    - Otherwise SECURE MVCMD is split into multiple PROCESS_BLOCK commands each sufficiently small to avoid clashing with loaded sections. *Note:* Each PROCESS_BLOCK command incurs a signature verification overhead so if the available buffer space is too small the boot time could be less than optimum.

As such, it is generally advisable to leave a region of memory free at the top of DDR which is slightly larger than the MVCMD image size. In most cases this is not a problem.

6.1 Secure Boot References

This chapter provides a brief overview of the secure boot feature and its key goal is to document the MDK build system targets which facilitate the generation of secure boot images. For full documentation of the Secure Boot process and eFUSE programming please see the following Movidius documents:

- MA2x5x Databook: Chapter 2.2 “Boot Operation”.
- MA2x5x Databook: Chapter 2.3 “Secure Boot”.
- Efuse Programming Application Note.

7 RTEMS

7.1 Overview

This chapter provides information about RTEMS integration into MDK, how to develop RTEMS applications on Myriad 2 and some guidelines on how to customize and extend the RTEMS version that is provided with MDK.

7.2 RTEMS features

RTEMS (Real-Time Executive for Multiprocessor Systems) is an open source fully featured Real Time Operating System that supports a variety of open standard application programming interfaces (API) and interface standards such as POSIX and BSD sockets. A list of RTEMS main features is presented below:

- POSIX 1003.1b API including threads.
- TCP/IP including BSD Sockets.
- Multitasking capabilities.
- Event-driven, priority-based pre-emptive scheduling.
- Responsive interrupt management.
- Dynamic memory allocation.
- High level of user configurability.
- Portable to many target environments.
- Support for many network protocols and file systems.

7.3 RTEMS in MDK

RTEMS is shipped as prebuilt libraries in the Tools:

```
[RTEMS dir] = [tools dir]/[tools version]/common/rtems/prebuilt/release
```

The Movidius RTEMS version for each shipped RTEMS exists in:

```
[RTEMS dir]/[SOC dir]/lib/include/myriad2_version.h
```

RTEMS tools	Rebased from	Official RTEMS branch	Official RTEMS version	Movidius RTEMS version
Older than 00.50.76.08	#2ca0a7 (11 Apr 2014)	master	4.10.99	<= v1.9
Newer than 00.50.76.08	#9c615b (4 Jan 2016)	4.11	4.11	>=v2.0
Newer than 00.80.0	#5bfeddc (21 Feb 2017)	master	4.12	>=v3.0
Newer than 00.88.0	#821acce (20 Sep 2017)	master	4.12	>=4.0

For more details of the specific changes in RTEMS, please see MDK Release Notes [ref13].

7.4 Writing RTEMS applications

Writing an RTEMS application is similar with writing a normal MDK application. The directory structure is the same as the one for normal applications. The only difference is the `MV_SOC_OS` variable which must be set inside the application Makefile. This variable is used to inform the build system that we are using RTEMS. By default it has the value “none “and must be set to “rtems” for RTEMS applications.

Example:

the Makefile of an RTEMS application must contain the following line:

```
…
# Ensure that the we are using the correct rtems libs etc
MV_SOC_OS = rtems
…
```

7.4.1 POSIX API

RTEMS implements a big part of POSIX 1003.1 standard. RTEMS supports a number of POSIX process, user, and group oriented routines in what is referred to as a "SUSP" (Single-User, Single Process) manner. RTEMS supports a single process, multithreaded POSIX 1003.1b environment. Even if only one process is used, RTEMS still provides process related routines with a dummy implementation to allow an easier port of applications coming from UNIX environment.

A complete reference of the parts of the POSIX standard implemented in RTEMS is available in the POSIX compliance guide [ref9] on RTEMS website.

The default entry point routine for POSIX applications is `POSIX_Init`(). It can be changed by defining `CONFIGURE_POSIX_INIT_THREAD_ENTRY_POINT` with a different value.

The maximum number of RTEMS objects needed by POSIX applications must be provided through configuration:

Example:

```
#define CONFIGURE_MAXIMUM_POSIX_THREADS           2
#define CONFIGURE_MAXIMUM_POSIX_SEMAPHORES        3
```

Trying to use a number of objects that is greater than the configured one will result in a runtime error.

NOTE: Beside the POSIX API there is also a classic RTEMS API. Some of the names for configuration macros is similar care should be taken not to use the wrong version. The POSIX macros always contain the word POSIX in their name. Example: `CONFIGURE_MAXIMUM_SEMAPHORES` should be used with classic API and `CONFIGURE_MAXIMUM_POSIX_SEMAPHORES` should be used with POSIX API.

7.4.2 Application configuration

For each application that uses RTEMS at least a minimal configuration should be provided. Configuration information can include the length of each clock tick, the maximum number of each RTEMS object that can be created, the application initialization tasks, and the device drivers in the application. This information is placed in data structures that are given to RTEMS at system initialization time.

Below is a configuration example for an RTEMS application using POSIX API:

```
#if !defined (__CONFIG__)
#define __CONFIG__
/* ask the system to generate a configuration table */
#define CONFIGURE_INIT
#define CONFIGURE_MICROSECONDS_PER_TICK 1000 /* 1 millisecond */
#define CONFIGURE_TICKS_PER_TIMESLICE 2 /* 2 millisecond */
```

```
#define RTEMS_POSIX_API
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER

#define CONFIGURE_POSIX_INIT_THREAD_TABLE

#define RTEMS_MINIMUM_STACK_SIZE 8192
#define CONFIGURE_MINIMUM_TASK_STACK_SIZE 2048

#define CONFIGURE_MAXIMUM_POSIX_THREADS         2
#define CONFIGURE_MAXIMUM_POSIX_SEMAPHORES      2

#include <rtems/confdefs.h>
#endif /* if defined CONFIG */
```

7.5 Working with threads

RTEMS provides multithreading support through POSIX API allowing thread creation, configuration and synchronization.

7.5.1 Thread configuration

Threads attributes can be configured by passing a `pthread_attr_t` structure to `pthread_create` function. c type can be used to control the scheduling policy. The members of `pthread_attr_t` must not be accessed directly. They should only be set and queried using functions provided by POSIX API.

Two commonly used scheduling policies implemented in RTEMS are:

- SCHED_FIFO
- SCHED_RR

`SCHED_FIFO` is a first in first out scheduling policy that runs each thread until completion before starting a new one.

`SCHED_RR` is a round robin policy with time slices. Each thread has an allocated time slice in which it can run. Once a thread is dispatched it will run until its allocated time expires or until it is blocked by some reason. When a thread finishes its execution another thread is dispatched and the procedure repeats.

Example of thread configuration:

```
void POSIX_Init ( void *args)
{
      //define a thread attribute structure
      pthread_attr_t attr;
       //thread attribute must be initialized before any changes are made to it
      if (pthread_attr_init(&attr) !=0) {
            printk( "pthread_attr_init error" );
            exit(1);
      }
      //set the schedule policy as explicit meaning that scheduling policy will
      //be set to the vales defined in attr parameter and not inherited from
      //the creating thread
      if (pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED) != 0) {
            printk( "pthread_attr_setinheritsched error" );
            exit(1);
      }
      //set the scheduling policy as round robin with time slicing
      if (pthread_attr_setschedpolicy(&attr, SCHED_RR) != 0) {
            printk( "pthread_attr_setschedpolicy error" );
```

```
            exit(1);
        }
        //use the above created thread attribute to transmit the attributes
        //at thread creation
        if (( rc =pthread_create( &thread1, &attr, &functionC,&counters[0]))) {
            printk( "Thread 1 creation failed: %d\n" , rc1);
            exit(1);
        }
         //other code can be placed here

        //wait for thread1 to finish it's execution
         if (!pthread_join(thread1, NULL)) {
            printk( "pthread_join error!" );
            exit(1);
        }
        exit(0);
}
```

7.5.2 Thread example applications

There are two example applications using POSIX threads in MDK.

- `mdk/examples/myriad2/Progressive/001_HelloWorld_RTEMS`: a simple application that runs two threads showing the way they thread switch happens in a time slice manner.
- `mdk/examples/myriad2/Progressive/002_HelloWorld_LOS-RTEMS_LRT`: an example that shows how to use leonOS to run RTEMS and leonRT for bare metal application.

7.5.3 Threads synchronization

When using events for intertasks communication, one application should use events ids from the available range, as defined in the following file:

`common/drivers/myriad2/socDrivers/leon/rtems/include/OsCommon.h`

7.6 RTEMS Build system

7.6.1 Overview

This chapter briefly describes the steps needed to build custom RTEMS using the provided sources from Movidius.

This guide assumes that you have already downloaded and extracted the RTEMS sources from either GIT or from movidius.org as tar archive.

7.6.2 Requirements

A Linux system (either 32- or 64-bit) with `autoconf`, GNU make and `gcc` installed. These steps were tested on:

- Make 3.8.5
- GCC 6.3.0
- Running in bash 4.3.11

If the `makefiles` of RTEMS have to be created (through the "bootstrap" application), the host has to match the following `automake` and `autoconf` versions:

- `Autoconf` == 2.69
- `Automake` == 1.12.6

7.6.3 Automated RTEMS build

- MDK contains a makefile that builds and installs Autotools, RTEMS-tools and RTEMS.
- Go to `mdk/common/utils/RTEMSbuild`.
- The makefile uses the following variables, which can get overridden, if needed:
  - `MV_TOOLS_DIR`, which point to tools to be used for build;
  - `MV_TOOLS_VERSION`, which point to tools version to be used for build;
  - `MV_COMMON_BASE`, which points to the MDK common folder.
- Run "`make all`".
- Upon successful completion the following folder/files are created:
  - StagingArea/logs: this contains build log files with build status data and xxx_complete files that indicate that the particular item (xxx) of the build has been completed, to rebuild a particular item of the build delete this file and run make again.
  - StagingArea/output: contains built libraries and include files.
  - StagingArea/tools: contains RTEMS tools and links to gcc tools used during build.
  - StagingArea/RTEMS/Build: contains built libraries and include files.
  - Staging Area/RTEMS/Tests: contains RTEMS test suites for each BSP built.

NOTE: To reduce build time or to specify a particular BSP for building, edit automate.sh and modify the RTEMS_BSP variable.

- The makefile supports a number of other targets for selectively building constituents of the overall build (built by "make all") they are:
  - "make rtems" – clone and build RTEMS only;

- "make rtems-debug" – clone and build RTEMS only, with the "—enable-rtems-debug" option;
- "make tests" – builds RTEMS test suite;
- "make auto-tools" – clone and build auto-tools;
- "make rtems-tools" – clone and bujld RTEMS tools.

- If there are any changes made to source files and an initial build was already completed, all that is required to build in the changes is:
  - Go to `mdk/common/utils/RTEMSbuild`
  - Delete `relevant_complete` file in `StagingArea/logs`.
  - Use the same settings for the environment variables.
  - Run "`make xxxxx`" where xxxxx is the relevant target from above.

7.6.3.1 RTEMS built with -enable-rtems-debug option

- Using make "rtems-debug" configures RTEMS with the `–enable-rtems-debug` option providing a heap-checker that can be used to find memory leaks and heap corruption. Please see RTEMS documentation for more detail.
- This option should be used for debug purpose only, remember to reconfigure RTEMS ("make rtems") for release purposes.

7.6.3.2 Using the new libraries

- The newly built RTEMS library and include files are located: "`StagingArea/output/myriad2-sparc-rtems/ma2x5x/lib`".
- Export these environment variables MV_RTEMS_LOS_LIB and MV_RTEMS_LRT_LIB to point to the above folder, MDK will then use these libraries in its build.

7.6.4 Manual RTEMS build

7.6.4.1 Building Autotools

The RTEMS build system uses the tools Automake and Autoconf that are provided by the Autotools package.

To build these tools:

- `git clone git@github.com:movidius/MovidiusRTEMS-source-builder.gitclone` the `MovidiusRTEMS-source-builder` repo (if you haven't already):
- `cd MovidiusRTEMS-source-builder/rtems cd [RTEMS-source-builder SRC dir]/rtems`
- `../source-builder/sb-check`
- `../source-builder/sb-set-builder --prefix=[TOOLS install dir]/4.12/rtems-autotools`

7.6.4.2 Building RTEMS-tools

RTEMS-tools is a collection of tools to help you use RTEMS.

To build these tools:

- `git clone git@github.com:movidius/MovidiusRTEMS-tools.git` the MovidiusRTEMS-tools repo (if you haven't already)
- `cd MovidiusRTEMS-tools cd [RTEMS-tools SRC dir]`
- `./waf distclean`

- `./waf configure --prefix=[TOOLS install dir]`
- `/waf build install`

7.6.4.3 Building RTEMS

If desired, two different repositories may be kept for different configurations on each Leon. Building RTEMS independently of the mdk convenience file is done with the following steps:

- The toolchain is using the prefix "`sparc-myriad-rtems`". For RTEMS, we need to provide a toolchain that uses a prefix "`myriad2-sparc-rtems`". For this reason, symbolic links or copies of the original toolchain files have to be created.
  - Create links from `sparc-myriad-rtems` files to `myriad2-sparc-rtems`:
    - `for i in \`ls [Toolchain dir]\`; do`
    - `ln -sf [Toolchain dir]/$i [Install dir]/bin/${i/sparc-myriad-rtems/myriad2-sparc-rtems};`
    - done.
  - Add the location of the links `[Install dir]/bin` to the PATH.
  - If your application requires any of the rtems-tools provided by the toolchain, the `[Tools directory]/common/rtems/bin` has to be added to the PATH.
- If `makefiles` have to be generated (usually once-off procedure):
  - `cd [RTEMS source dir]`
  - `./bootstrap -c`
  - `./bootstrap -p`
  - `./bootstrap`
- To configure a custom RTEMS build:
  - `cd [RTEMS build dir]`
  - `[RTEMS SRC dir]/configure -prefix=[Install dir] -target=myriad2-sparc-rtems --enable-drvmgr=no --enable-networking --enable-posix --enable-cxx -disable-tests -enable-rtemsbsp="ma2x5x"`
  - `If you wish to add extra debugging capability (heap-checker) you should add the --enable-rtemsp-debug option to the above configure command.`
- To build a custom RTEMS build:
  - `cd [RTEMS build dir]`
  - `make all RTEMS_BSP=[BSP]` (The allowed is BSP ma2x5x)
- To install a custom RTEMS build:
  - `cd [RTEMS build dir]`
  - `make install RTEMS_BSP=[BSP]`

Using the automated RTEMS build method (section 7.6.3) theses manual steps are executed automatically.

7.7 RTEMS benchmark suite

This section presents the results of the benchmark suite of RTEMS for Myriad 2. The purpose of the benchmark suite is to measure RTEMS's performance for several operations and scenarios. The benchmark suite is also used to compare different RTEMS versions of the same BSP.

The testsuite is run with L1 and L2 caches enabled, while all tests run from DDR. The results are evaluated for both LRT and LOS and present the number of cycles each operation takes to get executed.

- The test bellow measures the cycles the semaphore operations take to execute, with the assumption that they always execute without blocking/rescheduling the caller:

Test name(s)	DDR+L2C	LRT	LOS
tm01	rtems_semaphore_create: only case	2696	1165
	rtems_semaphore_delete: only case	2249	1386
	rtems_semaphore_obtain: available	111	111
	rtems_semaphore_obtain: not available NO_WAIT	111	111
	rtems_semaphore_release: no waiting tasks	115	115

- The tests bellow measure the cycles spent in "`rtems_semaphore_obtain`" in the scenario the caller tries to obtain an already taken semaphore and has to block:

Test name(s)	DDR+L2C	LRT	LOS
tm02 tm31 tm33 tm35	rtems_semaphore_obtain: counting/FIFO semaphore	3601	4
	rtems_semaphore_obtain: counting/priority semaphore	4370	2378
	rtems_semaphore_obtain: binary/FIFO semaphore	3873	2236
	rtems_semaphore_obtain: binary/priority semaphore	4618	2566

- These tests measure the cycles spent in "`rtems_semaphore_release`" in the scenario the caller releases a semaphore that unblocks another task, waiting for this semaphore:

Test name(s)	DDR+L2C	LRT	LOS
tm03 tm32 tm34 tm36	rtems_semaphore_release: counting/FIFO semaphore	2997	1127
	rtems_semaphore_release: counting/priority semaphore	48	24
	rtems_semaphore_release: binary/FIFO semaphore	3079	1209
	rtems_semaphore_release: binary/priority semaphore	51	27

- The tests bellow benchmark the operations related with task creation/execution/termination. The aim of the benchmark is to measure the overhead of each operation when interaction between tasks is required:

Test name(s)	DDR+L2C	LRT	LOS
tm04 tm05 tm06 tm07	rtems_task_restart: blocked task preempts caller	9649	5044
	rtems_task_restart: ready task -- preempts caller	7025	3293
	rtems_semaphore_release: task readied – returns to caller	1231	492
	rtems_task_create: only case	7455	3954
	rtems_task_start: only case	1384	703
	rtems_task_restart: suspended task – returns to caller	1827	764
	rtems_task_delete: suspended task	11096	4844
	rtems_task_restart: ready task -- returns to caller	1885	824
	rtems_task_restart: blocked task – returns to caller	2402	979
	rtems_task_delete: blocked task	10929	4202
	rtems_task_suspend: calling task	2814	1871
	rtems_task_resume: task readied preempts caller	2336	930
	rtems_task_restart: calling task	3151	2188
	rtems_task_suspend: returns to caller	662	301
	rtems_task_resume: task readied returns to caller	469	307
	rtems_task_delete: ready task	10778	5386
	rtems_task_restart: suspended task preempts caller	6248	3219

- The tests bellow benchmark operations related with task modes and clocks:

Test name(s)	DDR+L2C	LRT	LOS
tm08 tm17 tm18	rtems_task_set_priority: obtain current priority	134	132
	rtems_task_set_priority: returns to caller	637	389
	rtems_task_mode: obtain current mode	61	59
	rtems_task_mode: no reschedule	82	81
	rtems_task_mode: reschedule returns to caller	473	400
	rtems_task_mode: reschedule – preempts caller	3258	1665
	rtems_task_set_note: only case	220	175
	rtems_task_get_note: only case	178	180
	rtems_clock_set: only case	1340	1160
	rtems_clock_get_tod: only case	696	693
	rtems_task_set_priority: preempts caller	4027	2200
	rtems_task_delete: calling task	6308	4022

- The tests bellow benchmark the message queue related operations:

Test name(s)	DDR+L2C	LRT	LOS
tm09 tm10 tm11 tm12 tm13 tm14	rtems_message_queue_create: only case	9569	8178
	rtems_message_queue_send: no waiting tasks	213	211
	rtems_message_queue_urgent: no waiting tasks	216	215
	rtems_message_queue_receive: available	211	192
	rtems_message_queue_flush: no messages flushed	132	98
	rtems_message_queue_flush: messages flushed	125	117
	rtems_message_queue_delete: only case	3111	1720
	rtems_message_queue_receive: not available NO_WAIT	123	120
	rtems_message_queue_receive: not available caller blocks	4061	2471
	rtems_message_queue_send: task readied preempts caller	3339	1320
	rtems_message_queue_send: task readied returns to caller	1188	574
	rtems_message_queue_urgent: task readied preempts caller	3252	1318
	rtems_message_queue_urgent: task readied returns to caller	1194	572

- The tests bellow benchmark the functions related with RTEMS events:

Test name(s)	DDR+L2C	LRT	LOS
tm15 tm16	rtems_event_receive: obtain current events	65	63
	rtems_event_receive: not available NO_WAIT	83	84
	rtems_event_receive: not available caller blocks	3190	2033
	rtems_event_send: no task readied	86	83
	rtems_event_receive: available	815	122
	rtems_event_send: task readied returns to caller	1062	461
	rtems_event_send: task readied preempts caller	2702	1111

- The tests bellow benchmarks the signal related functions of RTEMS:

Test name(s)	DDR+L2C	LRT	LOS
tm19	rtems_signal_catch: only case	155	148
	rtems_signal_send: returns to caller	1472	915
	rtems_signal_send: signal to self	2007	1481
	rtems_signal: exit ASR overhead returns to calling task	698	556
	rtems_signal: exit ASR overhead returns to preempting task	2159	1669

- The tests bellow benchmarks the I/O related interfaces of RTEMS:

Test name(s)	DDR+L2C	LRT	LOS
tm20	rtems_partition_create: only case	3955	3034
	rtems_region_create: only case	3384	2461
	rtems_partition_get_buffer: available	551	534
	rtems_partition_get_buffer: not available	110	110
	rtems_partition_return_buffer: only case	668	500
	rtems_partition_delete: only case	1889	1072
	rtems_region_get_segment: available	1066	867
	rtems_region_get_segment: not available NO_WAIT	1202	736
	rtems_region_return_segment: no waiting tasks	750	663
	rtems_region_get_segment: not available caller blocks	6453	3595
	rtems_region_return_segment: task readied preempts caller	6748	2987
	rtems_region_return_segment: task readied returns to caller	2990	1369
	rtems_region_delete: only case	1701	984
	rtems_io_initialize: only case	45	41
	rtems_io_open: only case	35	32
	rtems_io_close: only case	35	34
	rtems_io_read: only case	33	22
	rtems_io_write: only case	34	34
	rtems_io_control: only case	34	34

- The tests bellow benchmarks the ident operations of various RTEMS interfaces:

Test name(s)	DDR+L2C	LRT	LOS
tm21	rtems_task_ident: only case	1217	939
	rtems_message_queue_ident: only case	1041	901
	rtems_semaphore_ident: only case	1283	988
	rtems_partition_ident: only case	1004	901
	rtems_region_ident: only case	1056	909
	rtems_port_ident: only case	955	905
	rtems_timer_ident: only case	1026	907
	rtems_rate_monotonic_ident: only case	1058	907

- The tests bellow benchmarks the operation rtems_message_queue_broadcast:

Test name(s)	DDR+L2C	LRT	LOS
tm22	rtems_message_queue_broadcast: task readied returns to caller	2679	1844
	rtems_message_queue_broadcast: no waiting tasks	125	122

Test name(s)	DDR+L2C	LRT	LOS
	rtems_message_queue_broadcast: task readied -- preempts caller	4028	2031

- The tests bellow benchmarks the operations related to the timer functionality of RTEMS:

Test name(s)	DDR+L2C	LRT	LOS
tm23	rtems_timer_create: only case	572	477
	rtems_timer_fire_after: inactive	518	505
	rtems_timer_fire_after: active	722	679
	rtems_timer_cancel: active	263	224
	rtems_timer_cancel: inactive	148	00
	rtems_timer_reset: inactive	479	457
	rtems_timer_reset: active	721	643
	rtems_timer_fire_when: inactive	1054	820
	rtems_timer_fire_when: active	1115	889
	rtems_timer_delete: active	848	657
	rtems_timer_delete: inactive	611	532
	rtems_task_wake_when: only case	3665	2202

- The tests bellow benchmarks the task_wake_after functionality of RTEMS, and its related function of announcing a tick has elapsed to a task, `rtems_clock_tick`:

Test name(s)	DDR+L2C	LRT	LOS
tm24 tm25	rtems_task_wake_after: yield returns to caller	292	273
	rtems_task_wake_after: yields preempts caller	2531	1621
	rtems_clock_tick: only case	3315	2394

- The test bellow benchmarks most of the internal functions of RTEMS related with context switching of threads and interrupt handling:

Test name(s)	DDR+L2C	LRT	LOS
tm26	rtems interrupt: _ISR_Disable	173	172
	rtems interrupt: _ISR_Flash	129	49
	rtems interrupt: _ISR_Enable	104	164
	rtems internal: _Thread_Disable_dispatch	1118	190
	rtems internal: _Thread_Enable_dispatch	62	62
	rtems internal: _Thread_Set_state	1230	671
	rtems internal: _Thread_Dispatch NO FP	1531	894
	rtems internal: context switch: no floating point contexts	1511	1145
	rtems internal: context switch: self	655	567

Test name(s)	DDR+L2C	LRT	LOS
	rtems internal: context switch to another task	239	121
	rtems internal: fp context switch restore 1st FP task	1334	747
	rtems internal: fp context switch save idle and restore initialized	645	696
	rtems internal: fp context switch save idle, restore idle	1762	1096
	rtems internal: fp context switch save initialized, restore initialized	128	118
	rtems internal: _Thread_Resume	1437	471
	rtems internal: _Thread_Unblock	448	263
	rtems internal: _Thread_Ready	282	257
	rtems internal: _Thread_Get	99	96
	rtems internal: _Semaphore_Get	74	72
	rtems internal: _Thread_Get: invalid id	41	41

- The test bellow benchmarks the overhead of interrupt handling:

Test name(s)	DDR+L2C	LRT	LOS
tm27	rtems interrupt: entry overhead returns to interrupted task	1379	1034
	rtems interrupt: exit overhead returns to interrupted task	644	540
	rtems interrupt: entry overhead returns to nested interrupt	244	230
	rtems interrupt: exit overhead returns to nested interrupt	244	104

- The test bellow benchmarks the functions of RTEMS related with ports:

Test name(s)	DDR+L2C	LRT	LOS
tm28	rtems_port_create: only case	1972	1342
	rtems_port_external_to_internal: only case	100	99
	rtems_port_internal_to_external: only case	97	97
	rtems_port_delete: only case	1549	953

- The test bellow benchmarks the rate monotonic functions of RTEMS:

Test name(s)	DDR+L2C	LRT	LOS
tm29	rtems_rate_monotonic_create: only case	2294	432
	rtems_rate_monotonic_period: initiate period returns to caller	3179	2147
	rtems_rate_monotonic_period: obtain status	431	111
	rtems_rate_monotonic_cancel: only case	1295	960
	rtems_rate_monotonic_delete: inactive	2155	1125
	rtems_rate_monotonic_delete: active	1158	666
	rtems_rate_monotonic_period: conclude periods caller blocks	3885	1579

- The test bellow benchmarks the barrier specific functions of RTEMS:

Test name(s)	DDR+L2C	LRT	LOS
tm30	rtems_barrier_create: only case	595	540
	rtems_barrier_ident: only case	1030	947
	rtems_barrier_delete: only case	688	597

The following tests benchmark too small operations, and there is no different in performance between LOS and LRT. Thus a single number will be presented for both cores.

- The test bellow benchmarks the accuracy of a timer. It first calculates the overhead of starting and immediately stopping a timer, and after that it calculates the time it takes to stop a timer after x LOOPS instructions. Each loop takes to execute 9 cycles. On the right column the average cycles per timer loop are calculated:

Test name(s)	DDR+L2C	LRT/LOS	cycles/loop
tmck	CCs for X LOOP, X={1k, 10k, 50k, 100k} :		
	Time Check: NULL timer stopped at	34	27
	Time Check: LOOP (1000) timer stopped at	9345	9377
	Time Check: LOOP (10000) timer stopped at	90236	90273
	Time Check: LOOP (50000) timer stopped at	450313	450202
	Time Check: LOOP (100000) timer stopped at	900062	900026

- The test bellow benchmarks the overhead in cycles of all internal RTEMS operations. The test is split in 4 segments, there is an idle loop between each segment to clean caches and internal RTEMS state that could influence the results:

Test name(s)	DDR+L2C	LRT/LOS
tmoverhd (1/4)	overhead: rtems_shutdown_executive	11
	overhead: rtems_task_create	9
	overhead: rtems_task_ident	11
	overhead: rtems_task_start	9
	overhead: rtems_task_restart	9
	overhead: rtems_task_delete	11
	overhead: rtems_task_suspend	9
	overhead: rtems_task_resume	9
	overhead: rtems_task_set_priority	11
	overhead: rtems_task_mode	11
	overhead: rtems_task_wake_when	9
	overhead: rtems_task_wake_after	11

Test name(s)	DDR+L2C	LRT/LOS
	overhead: rtems_interrupt_catch	11
	overhead: rtems_clock_get	9
	overhead: rtems_clock_set	9
	overhead: rtems_clock_tick	11

Test name(s)	DDR+L2C	LRT/LOS
tmoverhd 2/4	overhead: rtems_timer_create	10
	overhead: rtems_timer_delete	11
	overhead: rtems_timer_ident	9
	overhead: rtems_timer_fire_after	10
	overhead: rtems_timer_fire_when	12
	overhead: rtems_timer_reset	9
	overhead: rtems_timer_cancel	9
	overhead: rtems_semaphore_create	11
	overhead: rtems_semaphore_delete	9
	overhead: rtems_semaphore_ident	9
	overhead: rtems_semaphore_obtain	11
	overhead: rtems_semaphore_release	9
	overhead: rtems_message_queue_create	9
	overhead: rtems_message_queue_ident	11
	overhead: rtems_message_queue_delete	11
	overhead: rtems_message_queue_send	9
	overhead: rtems_message_queue_urgent	11
	overhead: rtems_message_queue_broadcast	11
	overhead: rtems_message_queue_receive	9
	overhead: rtems_message_queue_flush	9

Test name(s)	DDR+L2C	LRT/LOS
tmoverhd 3/4	overhead: rtems_event_send	9
	overhead: rtems_event_receive	11
	overhead: rtems_signal_catch	11
	overhead: rtems_signal_send	9
	overhead: rtems_partition_create	9
	overhead: rtems_partition_ident	11
	overhead: rtems_partition_delete	9
	overhead: rtems_partition_get_buffer	9

Test name(s)	DDR+L2C	LRT/LOS
	overhead: rtems_partition_return_buffer	11
	overhead: rtems_region_create	9
	overhead: rtems_region_ident	9
	overhead: rtems_region_delete	11
	overhead: rtems_region_get_segment	9
	overhead: rtems_region_return_segment	9
	overhead: rtems_port_create	11
	overhead: rtems_port_ident	11
	overhead: rtems_port_delete	9
	overhead: rtems_port_external_to_internal	11
	overhead: rtems_port_internal_to_external	11

Test name(s)	DDR+L2C	LRT/LOS
tmoverhd 4/4	overhead: rtems_io_initialize	11
	overhead: rtems_io_open	10
	overhead: rtems_io_close	9
	overhead: rtems_io_read	11
	overhead: rtems_io_write	11
	overhead: rtems_io_control	9
	overhead: rtems_fatal_error_occurred	9
	overhead: rtems_rate_monotonic_create	11
	overhead: rtems_rate_monotonic_ident	9
	overhead: rtems_rate_monotonic_delete	9
	overhead: rtems_rate_monotonic_cancel	11
	overhead: rtems_rate_monotonic_period	9
	overhead: rtems_multiprocessing_announce	9

7.8 External references

Supporting RTEMS documentation is available in addition to this guide and is referenced in chapter 12. These documents are referenced in this guide as [refX], where relevant.

8 MDK Software overview

8.1 Introduction

The MDK comprises common code which includes both drivers and components, and some example applications. This Section provides a brief description of the example applications and the re-usable components included in the MDK.

8.2 Example Applications

See the MDK-GettingStartedGuide.pdf and `mdk/examples/examples.html` for further programming examples and demonstration applications that ship with the MDK

8.3 Including test data into an application

8.3.1 Overview

This chapter describes one of the methods of how to embed raw data into our applications. Sparc-elf-objcopy is used to embed raw data into elf files.

8.3.2 How to

In order to include test data into the application, the method suggested to be used is the following. In the Makefile of the application we need to add a new object into the dependencies, using a predefined Makefile variable for this process, as follows:

```
RAWDATAOBJECTFILES + = $(DirAppObjDir) /testframe.o
```

This is then followed by the addition of the build rules for the newly added object:

```
$(DirAppObjDir) /testframe.o: $(MY_RESOURCE) Makefile
    $(ECHO) @mkdir -p $(dir $@)
    $(ECHO) $(OBJCOPY) -I binary $(REVERSE_BYTES) --rename-section .
data = $(DDR_DATA) \
   --redefine-sym  _binary_ $(subst /,_,$(subst .,_,$<)) _start
=inputFrame \
   -O elf32-sparc -B sparc $< $@,
```

where “ `DDR_DATA` ” and “ `REVERSE_BYTES` ” are defined as:

```
DDR_DATA = .ddr.data
REVERSE_BYTES =(option needed as a result of different endianness on
myriad1; for myriad2 we have “REVERSE_BYTES= “as memory areas have the
same endianness)
```

The user can either use one of the resources coming with the MDK package, or a specific internal data set, as follows:

```
MY_RESOURCE = (MV_EXTRA_DATA)
/data/common/still/80x60pYUV420p/DunLoghaire_80x60.yuv.
```

The symbol that is going to be used in the application is “inputFrame”, symbol which points to the raw data that is embedded into the sparc elf file. Therefore, on the LEON side of the application, we'll have:

```
extern u8 inputFrame;
```

NOTE: If the binary file inserted does not have a size that is at least multiple of 4 then the rule needs to be enhanced by using `--gap-fill` and `--pad-to` options to `objcopy` to ensure that the following sections will be aligned to 4, or else unaligned traps may occur at runtime since Leon may only access aligned data. For SHAVE only accessible information this does not matter, but if there are plans to share the structures, it is safer to adhere to this rule of alignment.

8.4 Intercommunication between processors

8.4.1 Overview

This chapter gives an insight of how `LeonOS`, `LeonRT` and the twelve `SHAVE`s can communicate with each other and what to keep track of in the applications developed.

8.4.2 Accessing Leon variables from SHAVE

If we want to share a variable declared in LeonOS core with a Shave the variable should be declared as follows:

```
int app#_myvar[4];
```

Where app is the name of the shave “mvlib” file declared in the Makefile and # is the number of the shave we want to share the variable with.

The way to access it from `SHAVE` is simply by name. In the `SHAVE` code one would have to:

```
extern int myvar [4];
```

And afterward the variable may just be used as any other one.

8.4.3 Accessing SHAVE variable from LEON

Let us assume the same variable myvar was declared in an application that was linked together into a “mvlib” file and this in turn placed over `shave3`. In this case, when accessing the variable from `LeonOS` one would need to use:

```
extern u32 someapp3_myvar [4];
```

And then use the variable `app3_myvar` in the code like any other variable.

8.4.4 Using unified symbols

It is sometimes desirable to use the same variable with the same physical address on multiple `SHAVE`s. This happens if a specific `SHAVE` function needs to be the same for all `SHAVE`s or if a specific variable needs to be shared.

This may be achieved through symbols unifying.

Let’s assume the previous myvar variable was declared in a `SHAVE` “mvlib” which was then loaded into both `shave3` and `shave4`. But there is no need to access:

```
extern u32 someapp3_myvar [4];
extern u32 someapp4_myvar [4];
```

Instead, a single variable in shared space should be used named myvar. This can be achieved through symbols unification.

This involves creating a `symuniq` file and using that in a rule to create a `shvXuniqlib` file. For this case, the `symuniq` file for shave3 would look like:

```
someapp3_myvar myvar
```

And the `symuniq` file for `shave4` like:

```
someapp4_myvar myvar
```

Please refer to the section describing Makefile rules for applications for more details on how to use this.

8.4.5 Using SHAVE entry point functions from the LEON

SHAVE entry point function serves as starting execution point on SHAVE. Their symbol is also accessible from the LEON side using:

```
extern u32 someapp0_entryPoint;
```

This can then be used to pass it to swcStartShave* functions like for example:

```
swcStartShave(0, ( u32 )&someapp0_entryPoint);
```

The following code sequence shows the declarations needed and the sequence of function calls required to start the execution of SHAVEs from the LEON side. Let's assume that we want to use 3 SHAVES – 0, 1, 2 in our application:

```
#define SHAVES_USED     3
extern u32 appName0_entryPoint;
extern u32 appName1_entryPoint;
extern u32 appName2_entryPoint;

u32 entryPoints[SHAVES_USED] = {
        ( u32 )&appName0_entryPoint,
        ( u32 )&appName1_entryPoint,
        ( u32 )&appName2_entryPoint
};

swcShaveUnit_t ShaveUsed[SHAVES_USED] = {0, 1, 2};

    for (i = 0; i < SHAVES_USED; i++)
    {
        swcResetShave(ShaveUsed[i]);
        swcSetAbsoluteDefaultStack(ShaveUsed[i]);
        //swcStartShaveCC(ShaveUsed[i], entryPoints[i], "ii" ,  ( u32
)&inBuffer[i], ( u32 )&outBuffer[i]);
    swcStartShave(ShaveUsed[i], entryPoints[i]);
     }
     swcWaitShaves(SHAVES_USED, ShaveUsed);
```

where SHAVES_USED is the number of SHAVEs used and extern u32 appName0_entryPoint is the SHAVE entry point function. Please refer to the section describing application Makefile rules f for more details on how to use this feature.

8.4.6 LeonOS – LeonRT

LeonRT entry point function serves as starting execution point on LeonRT. Its symbol is also accessible from the LeonOS side using:

```
extern u32 lrt_start;
```

This can then be used to take out the address of the starting point and pass it to DrvLeonRTStartup* function. Therefore, in the main code of the application, the following calls need to be done:

```
DrvLeonRTStartup((u32)&lrt_start); // Start the LeonRT application
DrvLeonRTWaitExecution();
```

In the Makefile of the application, the following variable needs to be associated with the:

```
LEON_RT_BUILD = yes
```

8.4.7 LeonOS – LeonRT – SHAVEs

In order to start LeonRT, the LeonRT entry point function has to be passed as a parameter to DrvLeonRTStartup on LeonOS. Its symbol is also accessible from the LeonOS side using:

```
extern u32 lrt_start;
```

This can then be used to take out the address of the starting point and pass it to DrvLeonRTStartup* function. Therefore, in the main code of the application, the following calls need to be done on the main code of the LeonOS:

```
DrvLeonRTStartup((u32)&lrt_start); // Start the LeonRT application
DrvLeonRTWaitExecution();
```

In the Makefile of the application, the following variable needs to be associated with the:

```
LEON_RT_BUILD = yes
```

On the LeonRT side, the SHAVE entry point is declared as follow:

```
extern u32 sampleApp0__entryPointName;
```

Function calls are as follow:

```
swcResetShave(SHAVE_NR);
swcSetAbsoluteDefaultStack(SHAVE_NR);
swcStartShave(SHAVE_NR, ( u32 )&sampleApp0__entryPointName);
swcWaitShave(SHAVE_NR);
```

For more information regarding the build process, creating the mvlib, consult section 5.5.2.

8.5 Synchronous vs. Asynchronous SHAVE running functions

8.5.1 Overview

This chapter aims to provide user an understanding of the synchronous and asynchronous SHAVE running functions.

8.5.2 Synchronous execution

In order to start the SHAVEs execution in a synchronous way, the user has the option of calling one of the following functions:

- **void swcStartShave(u32 shave_nr, u32 entry_point);**

It starts the non blocking execution of a SHAVE , having as parameters the SHAVE number and the entry point, that is a memory address to load in the SHAVE instruction pointer before starting.

It can be used in the following code sequence:

```
swcResetShave(SHAVE_NR);
 swcSetAbsoluteDefaultStack(SHAVE_NR); /// Function that sets absolute
stack address
swcStartShave(SHAVE_NR, (u32)&appNameSHAVE_NR_entryPoint);
swcWaitShave(SHAVE_NR); /// Function that waits for the shaves used to
finish
```

- **void swcStartShaveCC(u32 shave_num, u32 pc, const char *fmt, ...);**

It starts the non-blocking execution of a SHAVE, writes the value of a IRF/SRF/VRF registers from a specific SHAVE. It has the following parameters: the SHAVE number, the entry point function, a string containing i, s or v, according to irf, srf or vrf e.g. “iisv” and a variable number of parameters according to the entry point function parameter's number.

It can be used in the following code sequence:

```
swcResetShave(SHAVE_NR);
swcSetAbsoluteDefaultStack(SHAVE_NR); /// Function that sets absolute
stack address
swcStartShaveCC(SHAVE_NR, (u32)&appNameSHAVE_NR entryPoint, "ii" , (
u32 )&inBuffer, ( u32 )&outBuffer);//inBuffer and outBuffer are
parameters of the SHAVE entry point function
swcWaitShave(SHAVE_NR); /// Function that waits for the shaves used to
finish
```

It is important to keep in mind is that swcWaitShave () function needs to be called in order to wait for the SHAVE s execution to finish.

8.5.3 Asynchronous execution

In order to start the SHAVEs execution in an asynchronous way, the user has to make use of one of the following functions:

- **void swcStartShaveAsync(u32 shave_nr, u32 entry_point, irq_handler function);**

It starts the non blocking execution of a SHAVE , having as parameters the SHAVE number and the entry point, that is a memory address to load in the SHAVE instruction pointer before starting and the function to call when SHAVE finished execution.

It has to be used in the following code sequence:

```
swcResetShave(SHAVE_NR);
swcSetAbsoluteDefaultStack(SHAVE_NR);
swcStartShaveAsync(SHAVE_NR,(u32)&appNameSHAVE_NR_entryPoint,
(irq_handler)effectDone);
```

- **void swcStartShaveAsyncCC(u32 shave_num, u32 pc, irq_handler function, const char *fmt, ...);**

It starts the non-blocking execution of a SHAVE , writes the value of a IRF/SRF/VRF registers from a specific SHAVE . Having as parameters the SHAVE number and the entry point, that is a memory address to load in the SHAVE instruction pointer before starting, the function to call when SHAVE finished execution, a string containing i, s or v, according to irf, srf or vrf e.g. “iisv” and a variable number of parameters according to the number of parameters of the entry point function.

It can be used in the following code sequence:

```
swcResetShave(SHAVE_NR);

swcSetAbsoluteDefaultStack(SHAVE_NR);
swcStartShaveAsyncCC(SHAVE_NR,(u32)&appNameSHAVE_NR_entryPoint,
(irq_handler)effectDone, "ii", (u32)(&inBuffer), (u32)(&outBuffer));
```

8.6 Memory Manager

8.6.1 General description

The Memory Manager provides ways to dynamically allocate portions of CMX and DDR memory to applications at their request and free it for reuse when it is no longer needed. It assigns all the available CMX memory during initialization to itself and splits it into sections managing the correct distribution and handling of the memory. An additional section of DDR memory can be used for allocating memory. This section has to be defined by the user.

8.6.2 Requirements

- Execute manager from both Leons. The Memory Manager is not available to Shave code.
- Works on RTEMS and Bare-Metal.
- Preallocate configuration structures.
- Allocate/Deallocate functionality.
- Support queries/export of memory usage.
- Dynamic allocation rate support (smallest memory allocation unit).
- Allow even lower allocation rates inside memory units (MemMinAlloc).

8.6.3 Memory Manager API

8.6.3.1 Headers

The header to include in all applications that use the manager is "`memManagerApi.h`". Dependencies include "`mv_types.h`", "`mvMacros.h`" and "`DrvMutex.h`".

8.6.3.2 Data Types

```
typedef struct{
   unsigned int isInitialized;          /*manager initialized state*/
   unsigned int version;                /*structure version*/
   MemArea areas[MEM_TOTAL_COUNT];      /*areas of memory*/
}MemManag;

__attribute__((section(".ddr_direct.data"))) MemManag __MemManager;
```

`MemManag` is a data structure which is a direct representation of the memory, organizing data in memory areas the size of which is predefined. This structure can be exported externally to be parsed and processed. `__MemManager` is the structure used inside the manager component. It stands inside the ddr_direct.data section because its data needs to be shared between the two Leons. The `__MemManager` symbol has been uniquely defined inside the generic.mk file, so that the symbol bares the same name in both Leon OS and RT processors.

```
typedef struct{
   void* address;                       /*address of the memory
                                        area*/
   size_t size;                         /*total size of the
                                        current memory area*/
   size_t usage;                        /*size of allocated
```

```
                                                    memory inside this area*/
    size_t allocRate;                               /*size of smallest memory
                                                    allocation unit*/
    unsigned int nmbOfAllocs;                       /* number of allocators
                                                    inside current memory
                                                    area*/
    Allocator allocators[SYS_NMB_ALLOCS];           /*allocator units
                                                    structure*/
}MemArea;
```

SYS_NMB_ALLOCS is the maximum number of allocators available inside a memory area. It is defined in the header file. The default value is 256.

MemArea represents a memory area with predefined size used to structure individual sections of memory. It houses dynamically sized allocations inside it. Each allocation having a minimum allocation rate (allocRate).

```
typedef struct{
    void* address;                  /*address of the allocation*/
    size_t size;                    /*size of the allocator unit*/
    unsigned int state;             /*state of the
                                    allocator(free/used/static)*/
}Allocator;
```

Allocator is the memory portion allocated for specific usage. Its size is dynamic and occupies a certain number of allocation units inside the memory area.

```
typedef enum{
    NO_ERROR = 0,                   /*Returned whenever operations
                                    are successful*/
    MEMALLOC_INVALID_SIZE = 10,     /*Invalid size value*/
    MEMALLOC_SIZE_EXCEEDED = 11,    /*Size bigger than available
                                    space*/
    MEM_INVALID_ADDRESS = 12,       /*Invalid address used*/
    MEM_EXPORT_ALLOCMEM = 13,       /*Allocate more memory for
                                    export buffer*/
}MemMgrErrorCode;
```

MemErrorCode is an error enumerator for the memory manager system. More error codes can be added and used by APIs to return a specific error code.

```
typedef enum{
    CMX_AREA0 = 0,
    CMX_AREA1 = 1,
    …
    CMX_TOTAL_COUNT,
```

CMX_TOTAL_COUNT represents the total number of areas the CMX memory is split into. It is defined in the header file.

MEM_TOTAL_COUNT is described as the total number of memory areas the memory

```
    DDR_AREA = CMX_TOTAL_COUNT,
    MEM_TOTAL_COUNT
}MemMgrAreas;
```

manager is handling. It's composed by the CMX areas and the DDR area.

`MemMgrAreas` enumerates all the memory areas the Memory Manager can handle, consisting of both CMX slices and DDR area.

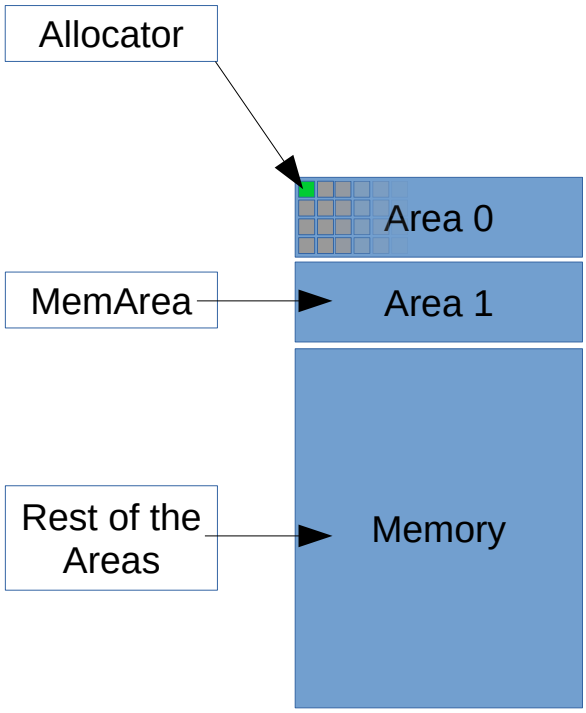


Figure 9: Data types illustration on a memory block

8.6.3.2.1 Memory consumption

The Memory Management data structure (`MemManag`) will rest inside the DDR memory of the system in order to save CMX memory space. This structure will occupy approximately 40.2Kb of ram memory when dividing the CMX memory block into 12 areas (ma2x5x), or 52,5Kb when CMX block has 16 areas (ma2x8x).

8.6.3.3 Functions

- `void MemMgrInitialize();`

Allocate memory slices into CMX memory defining the structure that describes the memory. Different configurations may be created by users. For a default Myriad application CMX slices will be mapped to specific areas each. This function will also initialize a section of the DDR memory if the user defines the `MEM_MGR_DDR_SIZE` symbol in the application's makefile.

```
void* MemMgrAlloc(size_t size, MemMgrAreas area);
```

Allocate memory equal to the specified size inside the given memory area. Allocations will always be rounded up to the MemManager's allocation rate. The function will return the allocation's address or NULL if allocating isn't possible.

➢ `MemMgrErrorCode MemMgrFree(void* address);`

Free up memory previously allocated by the memory manager located at the inputted address. If address is invalid, *MEM_INVALID_ADDRESS* error will be returned.

➢ `MemMgrErrorCode MemMgrBufferExport(size_t bufSize, char *csvBuffer, size_t *csvLenght);`

Get the memory's current state as a buffer. Data is structured in a .csv style format so it can easily be exported. The user must input a buffer to the function to be filled with data from the memory manager. If the buffer is not big enough, the error `MEM_EXPORT_ALLOCMEM` will be returned.

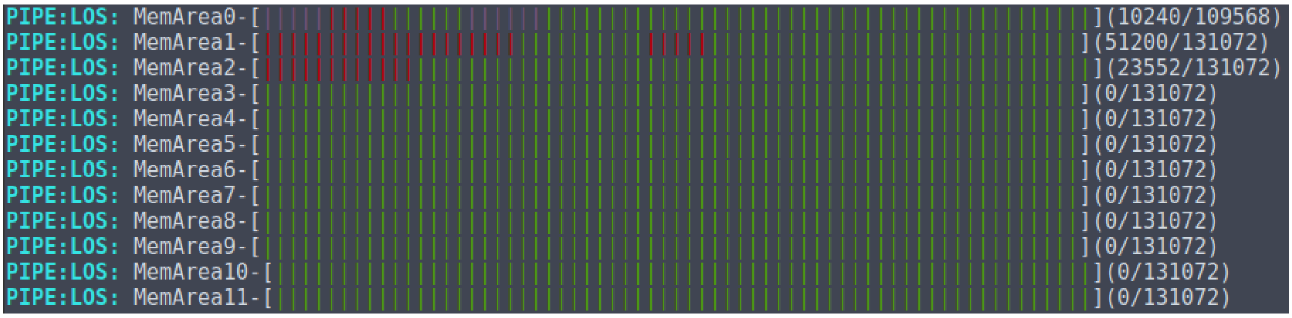
Example of usage:

```
 char csvExport[9999];
 size_t csvLenght;
 error = MemMgrBufferExport(9999, csvExport, &csvLenght);
 if (error == NO_ERROR)
   //Export memory structure to .csv file (using VcsHooks!)
     saveMemoryToFile((u32)&csvExport, csvLenght, "output/export.csv");

   // OR run command "savefile exportName.csv csvExport 7200" while running in
debug mode
```

➢ `void MemMgrExportDraw();`

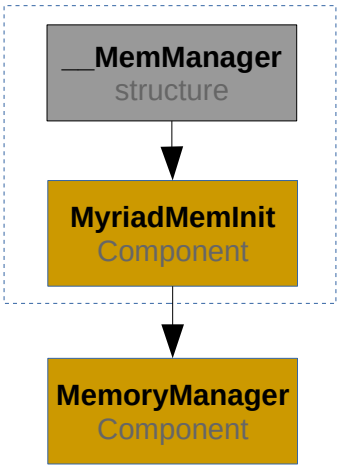
Graphically illustrate the current state of the memory mapped inside the memory manager using the console print.



➢ `void* MemMinAlloc(Allocator* AllocHandle, unsigned int size);`

Allocate data smaller than a single allocator unit inside the memory. This `MemMinAlloc` will be based on the standard malloc code, only allocating inside the allocator pointed to by handle. Once used with `MemMinAlloc`, the user may no longer use this allocator in other ways.

8.6.4 Memory Manager data structure



The memory structure the manager uses is called `__MemManager`. Using this name is mandatory as it is inserted into the MDK's generic.mk as a unique symbol.

This structure is defined inside the `MyriadMemInit` header file and it is initialized to default values for the myriad chip inside the same component when calling the `MemMgrInitialize()` function. If the user wants he can create a custom initialization function/component, the only thing that needs to remain the same is the structure's name and members.

After initializing, the `__MemManager` structure can be used by the `MemoryManager` component. It will have all the memory areas used for allocating memory defined inside it.

8.6.5 Debugging

The Memory Manager has multiple logging keywords inserted into the code. It uses the "`mvLog.h`" in order to do so. `MvLog` has logging capabilities over simple `printf` and allows 5 different logging levels. In order to turn on logging, the user must include "`mvLog.h`" into the application and set the default logging level by using the `mvLogDefaultLevelSet(debugLevel)` function.

The 5 different logging levels:
** MVLOG_DEBUG = 0*
** MVLOG_INFO = 1*
** MVLOG_WARN = 2*
** MVLOG_ERROR = 3*
** MVLOG_FATAL = 4*

The debug logging keywords inside the memory manager use the `MVLOG_DEBUG` debug level. So in order to see the debug logging the application must set the level to 0.

`MVLOG_INFO` debug level is used to show process information in certain functions.

Whenever an error occurs inside the memory manager, the `mvLog` will log it as a `MVLOG_ERROR` debug level.

NOTE: By setting the lowest debug level(0), all information will be printed.

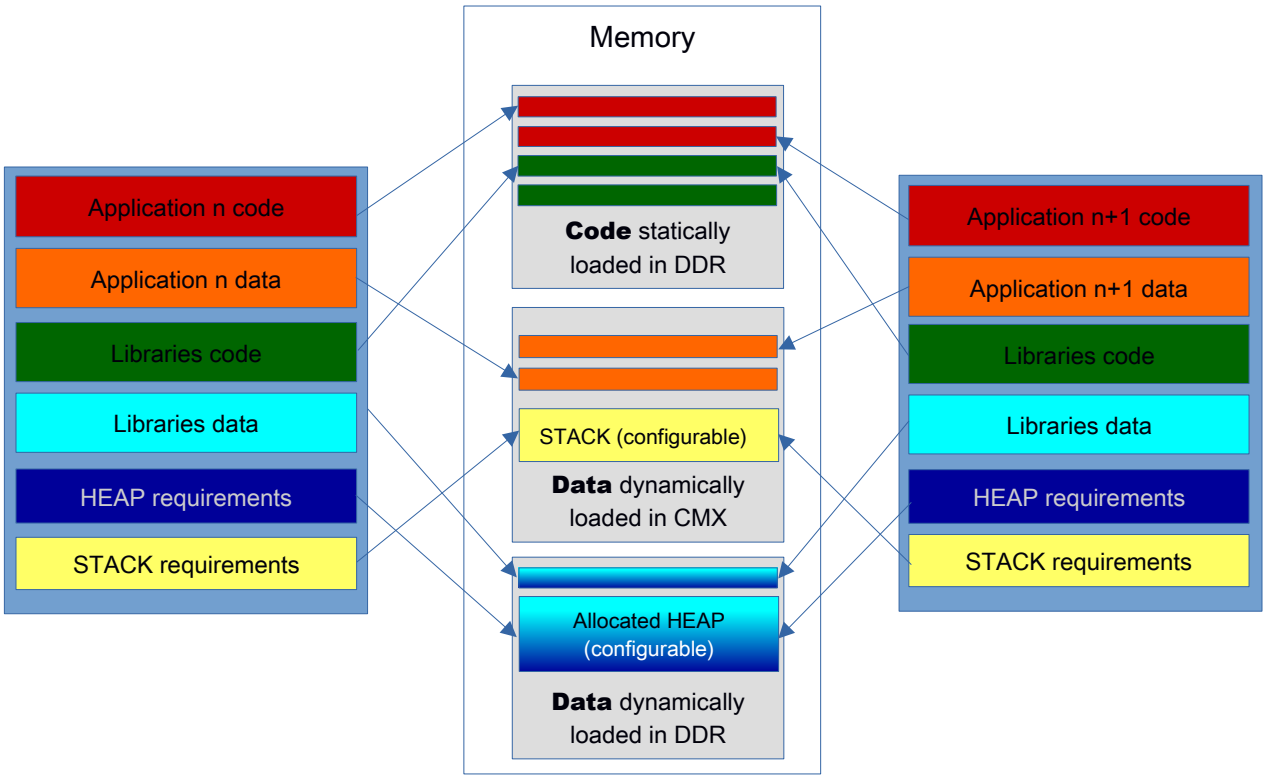
`MemMgrExportDraw()` is very useful for debugging the memory manager because it illustrates the memory's state at the moment when the function is called in the application.

8.7 Mixed Instantiation Applications

8.7.1 Overview

Mixed Instantiation Application infrastructure allows runtime loading of application libraries on shave processors.

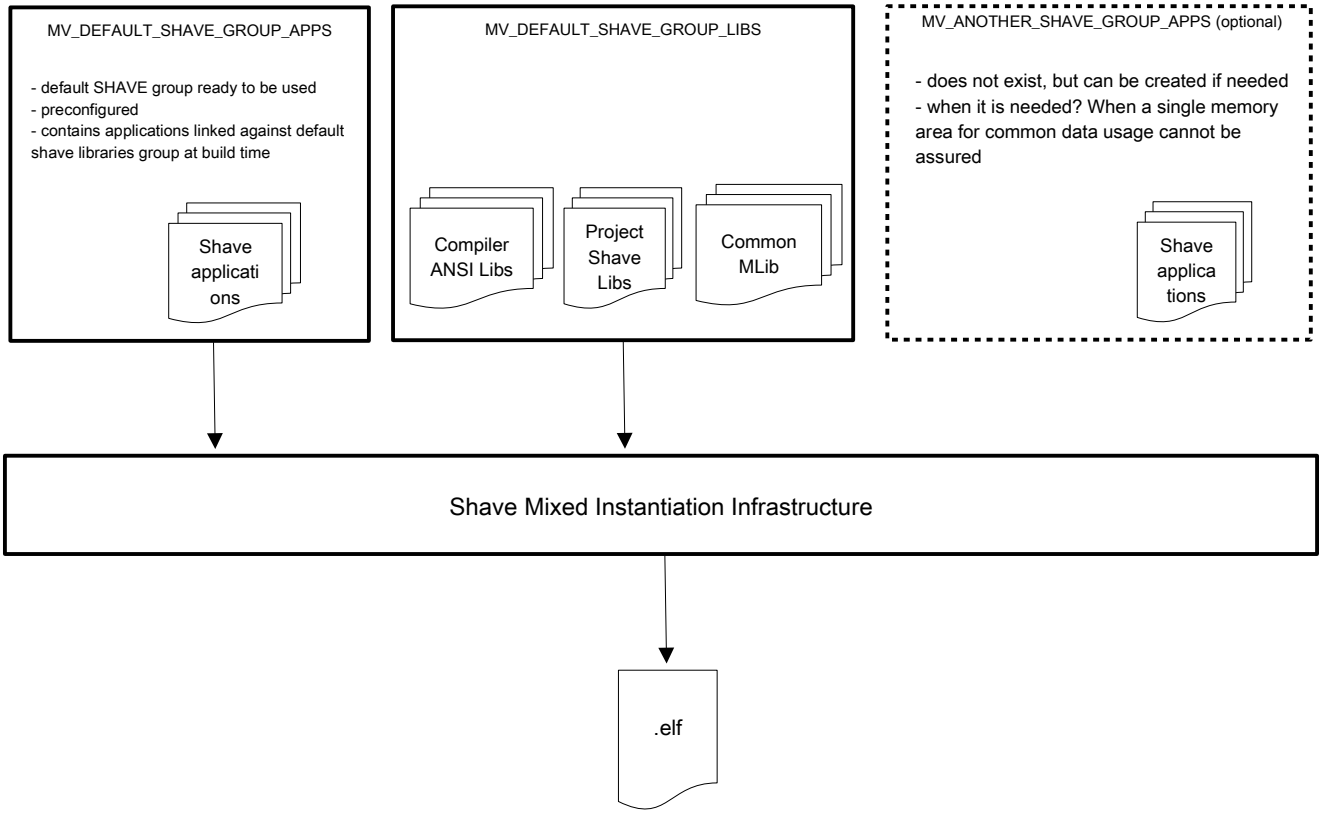
The benefits of runtime loading, beside a greater flexibility, include optimal usage of CMX for data requirements, where performance is required and moving most of the instruction code to DDR to take advantage of the caching infrastructure. The following picture illustrates the memory handling concept when multiple user applications are involved.



To achieve the memory requirements of various applications, the Mixed Instantiation Application operates with GROUPS. A group is a collection of applications and libraries which target common memory requirements.

Although the above diagram shows library data meant to stay in CMX, by default, this data will be placed in DDR in order to account for data requirements coming from libraries being prohibitively big to fit in CMX.

The diagram below presents the simplified concept of Mixed Instantiation Application.



The diagram above mentions the existence of SHAVE library groups. The reason for supporting multiple library groups may not be obvious at first sight. In classic architectures, there is usually a single library group. However, in the case of this paradigm, a group of library will implicitly create a requirement for the group's data requests. In some cases this may be very big and then it needs to be used in DDR but CMX would be preferred if possible. In other applications the group needs may be smaller so, it is favorable to create different groups for these situations. More about groups may be found in section 8.7.4.2, Starting the new group.

8.7.2 Usage

Mixed Instantiation Applications infrastructure is available starting with MDK release version 16.06. However, in previous releases this infrastructure was called Shave Dynamic Loading.

All examples that use SHAVE's demonstrate the existing features of Mixed Instantiation Applications for shave's. One of the simplest examples that demonstrates the usage can be found in MDK under `../examples/Progressive/ma2x5x/001_HelloWorldShave`.

Under folder `../shaveDynApps`, the user can add all the shave applications that need to be used dynamically. The Mixed Instantiation feature is enabled via configuration settings in the Makefile.

8.7.2.1 Notes on API for launching dynamic infrastructure apps

The purpose of this section is to introduce the helper APIs available for running Mixed Instantiation infrastructure applications.

8.7.2.1.1 swcSetupDynShaveApps / OsDrvSvuSetupDynShaveApps

This function allocates the necessary memory for all configured instances of an application. It also performs some checks to ensure shave allocation is valid.

It must be called prior to using swcRunShaveAlgo()/OsDrvSvuRunShaveAlgo() and can be used from both Leon OS and Leon RT.

The function parameter `svuList` is a pointer of `swcShaveUnit_t` type, which will be copied in the internal array `instancesData->shaveList[TOTAL_NUM_SHAVES]` of the dynamic shave application context variable because this information is required in subsequent dynamic functions until the final `swcCleanupDynShaveApps()` / `OsDrvCleanupDynShaveApps()` call.

This function also has an RTEMS OS version: OsDrvSvuSetupDynShaveApps.

8.7.2.1.2 OsDrvSvuOpenShaves

This function creates a handler and allocates an RTEMS semaphore for every shave in the shave list. It will also mark the shave as taken, preventing its allocation for other shave applications.

The function has the following parameters:

- `parameter swcShaveUnit_t * svuList`, points to an array variable containing a list of shaves which will be reserved for this shave application.
- `parameter uint32_t shavesInList`, which specifies the number of shaves in the list.
- `parameter protection`, is an enum containing types of protection for driver handlers and it is defined as `OS_MYRIAD_DRIVER_PROTECTION` in `OsCommon.h`

8.7.2.1.3 swcRunShaveAlgo / OsDrvSvuRunShaveAlgo

This function sets up and launches one dynamic application instance on a certain shave. The target shave is automatically selected from a group of shaves preassigned by the user via the swcSetupDynShaveApps() function.

It allocates all the necessary memory to ensure that the shave application instance can run, then it loads the dynamic library, and then it starts the shave. It is mandatory to configure the list of shaves to be used (via swcSetupDynShaveApps) prior to calling this function.

It can be used from both Leon OS and Leon RT.

This function has also an RTEMS OS version: `OsDrvSvuRunShaveAlgo.`

8.7.2.1.4 swcRunShaveAlgoCC / OsDrvSvuRunShaveAlgoCC

Similar to swcRunShaveAlgo / OsDrvSvuRunShaveAlgo, with the option to use parameters for shave's entry point.

8.7.2.1.5 swcRunShaveAlgoOnAssignedShave / OsDrvSvuRunShaveAlgoOnAssignedShave

This function sets up and launches one dynamic application instance on a specifically requested shave. Checks if the requested shave is not running and if it's part of the list of shaves that the dynamic shave application was set up with via function `swcSetupDynShaveApps()` and returns the error `MYR_DYN_INFR_SHAVE_BUSY_OR_INVALID`, if it's not.

It allocates all necessary memory, loads the dynamic library, then starts the shave.

It can be used from both Leon OS and Leon RT.

In the case of a multi-threaded usage scenario it is recommended to use this function to run the dynamic shave application on the shaves assigned for the current thread. The dynamic shave application context variable `instancesData→shaveList[TOTAL_NUM_SHAVES]` now accepts multiple svuList's assigned to it, but it doesn't store any information about the threads they had been assigned from. If multiple shaves

need to be started from different threads, it is recommended to call this function from each thread for the shaves that should be started by that thread in order to prevent starting the same shave from multiple threads.

This function also has an RTEMS OS version: OsDrvSvuRunShaveAlgoOnAssignedShave.

8.7.2.1.6 swcRunShaveAlgoOnAssignedShaveCC / OsDrvSvuRunShaveAlgoOnAssignedShaveCC

Similar to swcRunShaveAlgoOnAssignedShave / OsDrvSvuRunShaveAlgoOnAssignedShave, with the option to use parameters for shave's entry point.

8.7.2.1.7 swcCleanupDynShaveApps / OsDrvSvuCleanupDynShaveApps

This function frees previously allocated memory for all configured instances of one application.

It can be called after usage of swcRunShaveAlgo() / OsDrvSvuRunShaveAlgo(). Can be used from both Leon OS and Leon RT.

This function also has an RTEMS OS version: OsDrvSvuCleanupDynShaveApps.

8.7.2.1.8 swcDynStartShave

This function is similar to swcStartShave, but has a Module Data parameter instead of an entry point. Module Data configuration is shave application specific and contains, among others, a reference to its entry point. A notable difference, relevant to dynamic loading infrastructure, is that it will reach the shave's entry point in two stages. First, a master entry point will be entered where pre-start operations take place (`DEFAULT_SHAVE_GROUP____AllTimeEntryPoint`). One example would be the heap memory initialization, which needs to be done before executing any shave code. Only after this phase the program will jump in the application's entry point.

Please also check some important aspects, when dealing with an additional group, in chapter 8.7.4.2 Starting the new group.

8.7.2.1.9 OsDrvSvuCloseShaves

This function releases the resources allocated for the given list of shaves. It does not substitute the role of OsDrvSvuCleanupDynShaveApps function.

8.7.2.1.10 SwcRequestUnallocatedShaves / OsDrvRequestUnallocatedShaves

This function has one input paramter of type `u32`, and one output parameter of type `swcShaveUnit_t *`.

As input parameter it takes the number of requested unallocated shave applications (u32), and as the output parameter it takes the svuList array (swcShaveUnit_t *) in which it will store the found unallocated shaves for the application. If it is not able to find the number of requested shaves then it will return the error code: `MYR_DYN_INFR_NUMBER_OF_FREE_SHAVE_NOT_AVAILABLE` as return value.

This function also has an RTEMS OS version: OsDrvRequestUnallocatedShaves.

8.7.2.1.11 SwcCleanupDynShaveListApps / OsDrvSvuCleanupDynShaveListApps

This function takes one input/output parameter of `(DynamicContext_t *)` type and two input parameters of types `(swcShaveUnit_t *)` and `(uint32_t)`.

This function frees the memory allocated for the dynamic shave application data and heap and updates the context variable with the changes. It takes 2 input variables: the list of shaves (`svuList`) that the dynamic shave application should be run on and the number of elements in this list.

It should be called after OsDrvSvuCloseShaves. Can be used from both Leon OS and Leon RT.

This function also has an RTEMS OS version: OsDrvSvuCleanupDynShaveListApps.

8.7.2.1.12 SwcGetFreeShaveNumber / OsDrvGetUnallocatedShavesNumber

This function returns the number of unallocated shaves to any application.

This function also has an RTEMS OS version: OsDrvGetUnallocatedShavesNumber.

8.7.2.1.13 swcSetNewHeapLocation

This function frees the memory allocated for the dynamic shave application heap and updates the context variable with the new address of the heap desired location.

This function takes one input/output parameter of `(DynamicContext_t *)` type and two input parameters of types `(unsigned char *)` and `(swcShaveUnit_t)`.

8.7.2.1.14 swcSetNewAppDynDataLocation

This function frees the memory allocated for the dynamic shave application data and updates the context variable with the new address of the application data desired location.

This function takes one input/output parameter of `(DynamicContext_t *)` type and two input parameters of types `(unsigned char *)` and `(swcShaveUnit_t)`.

8.7.2.1.15 swcSetGrpDynDataLocation

This function frees the memory allocated for the dynamic shave group data and updates the context variable with the new address of the group data desired location.

This function takes one input/output parameter of `(DynamicContext_t *)` type and two input parameters of types `(unsigned char *)` and `(swcShaveUnit_t)`.

8.7.3 Basic Build Configuration

To use Mixed Instantiation Applications the user must perform the following steps:

1. **Create the file structure**

   See MDK examples `../examples/HowTo/DynamicInfrastructure_C/CPP` for a demonstration. Most of the rules for creating applications loading the data dynamically are similar to previously existing general MDK methods.

   If all shave applications are placed under a folder named shaveDynApps, then these will be auto-detected and steps 2 to 4 are not necessary. In this case, each shave application name will be built from its folder name.

   If the above folder structure is not used, declaring a few applications could be done similar to:

2. **Configure shave applications in make file**

```
#------[ Local shave applications sources ]-----#
# Choosing C sources for MesgOne application on shave

MesgOneApp = shave/MesgOne/MesgOne
SHAVE_C_SOURCES_MesgOne =  $(wildcard $(DirAppRoot)/shave/MesgOne/*.c)

# Generating list of required generated assembly files for the MesgOne
app on shave
```

```
SHAVE_GENASMS_MesgOne = $(patsubst %.c,$(DirAppObjBase)%.asmgen,$
(SHAVE_C_SOURCES_MesgOne))

# Generating required objects list from sources
SHAVE_MesgOne_OBJS = $(patsubst $(DirAppObjBase)%.asmgen,$
(DirAppObjBase)%_shave.o,$(SHAVE_GENASMS_MesgOne)) $(patsubst %.asm,$
(DirAppObjBase)%_shave.o,$(SHAVE_ASM_SOURCES_MesgOne))
PROJECTCLEAN += $(SHAVE_GENASMS_MesgOne) $(SHAVE_MesgOne_OBJS)

# Choosing C sources for MesgTwo application on shave
MesgTwoApp = shave/MesgTwo/MesgTwo
SHAVE_C_SOURCES_MesgTwo =  $(wildcard $(DirAppRoot)/shave/MesgTwo/*.c
```

3. Subscribe to a SHAVE applications group

Each shave application must belong to a group. To see more details about group usage, check section 8.7.4, Advanced Build Configuration (optional).

The MDK has a predefined default group for convenience : `MV_DEFAULT_SHAVE _GROUP_APPS`. This group includes the default SHAVE C/C++ libraries: ISO C libs, MDK generated applications: swCommon.mlibv and libraries created from SHAVE_COMPONENTS if SHAVE_COMPONENTS=yes. The user can start adding applications to this group.

To do this, add each shave application, after they were defined, but before including `generic.mk`

```
# ------[ Tools overrides ]-----#
# Add SHAVE applications here which you would wish to link against the
default shave
# libraries group
# The default shave library group contains all ANSI C functionality as
described in the # moviCompile manual as well as common MDK libraries/
shave drivers

MV_DEFAULT_SHAVE_GROUP_APPS := $(MesgOneApp) $(MesgTwoApp) $
(MesgThreeApp)
...
# Include the generic Makefile
include $(MV_COMMON_BASE)/generic.mk
...
```

4. Configure build rules for shave applications

This is done almost in the same way as all other SHAVE applications. The one difference we need to point out is that there is no need to specify any of the application's group libraries in the link path. For example, since the DEFAULT group includes the ANSI libraries, there is no more need to pass these to the mvlib rules:

```
#-------[ Local shave applications build rules ]-----#

# Describe the rules for building MesgOneApp and MesgTwoApp
applications. Simple rule
# specifying which objects build up the said application.
# Each application will be built into a library.

ENTRYPOINTS1 = -e $(MesgOne_MainEntry) --gc-sections
```

```
$(MesgOneApp).mvlib : $(SHAVE_MesgOne_OBJS)
   $(ECHO) $(LD) $(ENTRYPOINTS1) $(MVLIBOPT) $(SHAVE_MesgOne_OBJS) -o
$@

ENTRYPOINTS2 = -e $(MesgTwo_MainEntry) --gc-sections
$(MesgTwoApp).mvlib : $(SHAVE_MesgTwo_OBJS)
   $(ECHO) $(LD) $(ENTRYPOINTS2) $(MVLIBOPT) $(SHAVE_MesgTwo_OBJS) -o
$@
```

Shave entry point (`MesgOne_MainEntry`) is part of module configuration data. Configuration data already has default values and it's ready to be used, but can be overridden at runtime if needed.

Default configuration for shave entry point name is Entry.

For details about module configuration data see chapter 8.7.5, Module data configuration.

5. Configure heap and stack size

Like the entry point name, shave heap and stack sizes are part of module configuration. They have default values configured, but the user is able to overwrite their size and position at runtime, if necessary. Also, their location can be changed.

To change statically default heap and stack configuration for each shave application, it is sufficient to pass the heap/stack sizes individually for each application in the local Makefile before the inclusion of `generic.mk`. For example:

```
#------[ Shave applications section ]-----#
#Some overrides for MesgOne
MesgOne_HEAPSIZE  = 10*1024
MesgOne_STACKSIZE = 4*1024
```

More details in section 8.7.5, Module data configuration.

6. Configure stack instrumentation for shave applications

moviCompile provides 2 shave application stack instrumentation functions:

a. Stack-overflow instrumentation;

b. Stack-usage instrumentation.

These options are documented in detail in the moviCompile documentation in the chapter named "Stack Instrumentation". In order to activate this at compile time, the corresponding command line arguments should be added to the MVCCOP build system variable.

If enabled at build time, the MDK built-in `swcWaitShave`, `swcWaitShaves`, `OsDrvSvuRunShave` and `OsDrvSvuWaitShaves` functions will report an error if the stack overflow has been identified as the shave application's exit code:

a. `MYR_DYN_INFR_ERROR_STACK_OVERFLOW` for BM.

b. `OS_MYR_DYN_INFR_ERROR_STACK_OVERFLOW` for RTEMS.

7. Enable Application name information used during debugging

A new build system variable DEBUGLOADCONTEXT had been introduced that if set to the value “YES” can enable a debug context information to be added for each mixed instantiation shave applications running.

If this functionality is enabled the application group data library and application elf file name excluding the file extension will be stored in memory right before the location of these elf files. This information can be used during debugging to identify the location of the code section for each running shave application.

By default, the variable is set to NO from the build system and can be overwritten in the application Makefile.

The application name information has the following format:

- first 4 bytes will contain length of the name,
- followed by the name of the application that the code and data segment belongs to.
- the actual ELF content aligned at a memory address which is some multiple of 4 bytes.

The application group data library name information has the following format:

- first 4 bytes will contain length of the name,
- followed by the name of the application group data library that the data segment belongs to.
- the actual ELF content aligned at a memory address which is aligned to a multiple of 4 bytes.

During runtime, before starting the shave processor for a registered shave application, the name of the application and application group data library will be stored in the following global context variables:

- __ShvXdataContextLoaded[shave#] (it will store the name of the application loaded),
- __ShvZdataContextLoaded[shave#] (it will store the name of the group by wich this application belongs).

In order to see the name of the applications loaded, at any given particular moment and on any of the shaves, the following command should be run in debug mode:

```
for { set i 0} { $i < 12 } {incr i } { puts [format "%02d %s" $i [mget -asciiz <name_global_variable>\[$i\]]] }
```

where `<name_global_variable>` can be `__ShvXdataContextLoaded` or `__ShvZdataContextLoaded`.

8. Enable allocation with the MemoryManager component

In order to enable the allocation of all the memory locations needed by shave application and group with the Memory Manager component, a new build option was added.

In order to enable this information, the build system variable `SHAVE_APP_ENABLE_MEMORY_MANAGER_ALLOCATION` needs to be set to the value `YES`. By default, this variable is set to `NO` from the build system and can be overwritten in the application Makefile.

Enabling this functionality allows the memory allocations for the shave application to not be in the Leon HEAP memory, but to be made in the available memory of the CMX shave slice.

If the CMX shave slice doesn’t have enough free memory available then the MISA FW automatically allocates the necessary memory in DDR.

If custom memory locations are desired then it is possible to use the functions `swcSetNewHeapLocation`, `swcSetNewAppDynDataLocation`, and `swcSetGrpDynDataLocation` to free the allocations and to assign a new address for these memory locations from the LEON side before starting the shave application. In this case the application must allocate the memory for the locations and must take care to deallocate it when not necessary. These memory locations will be initialized by the MISA FW by memcopy-ing the elf file content in the assigned memory location.

9. Build

No extra build parameters are necessary.

8.7.4 Advanced Build Configuration (optional)

8.7.4.1 Creating a new group

Each shave application must belong to a group. Each group of shave applications has a corresponding library group. Shave applications are linked against their library group.

MDK comes with a predefined and preconfigured shave group, `MV_DEFAULT_SHAVE_GROUP_APPS`, and its library group, `DEFAULT_SHAVE_GROUP_LIBS`.

It is possible to encounter situations where memory requirements of all shave applications cannot be fulfilled by the single usage of the default shave group and default shave libs. In this case, the user has the ability to create an additional group.

To do this, in your application's `Makefile` create another group and name it `MV_YOURGROUPNAME_SHAVE_GROUP_APPS`.

```
MV_DEFAULT_SHAVE_GROUP_APPS := $(MesgOneApp) $(MesgTwoApp) $
(MesgThreeApp)
MV_YOURGROUPNAME_SHAVE_GROUP_APPS := $(MesgFourApp) $(MesgFiveApp)
```

Then add your group to `MV_SHAVE_GROUPS`, in your local `Makefile` before the inclusion of `generic.mk`, just like `DEFAULT_SHAVE_GROUP`:

```
MV_SHAVE_GROUPS += YOURGROUPNAME_SHAVE_GROUP
```

Finally, the new group can have some libraries associated. This line is also located in your local application `Makefile` before the inclusion of `generic.mk` file.

```
# Default group is slightly special because we include all default
libs
# default rule already brought the CommonMlibFile in so we must take
it out when
# including the libraries

YOURGROUPNAME_SHAVE_GROUP_LIBS := $(YourLibrary1) $(YourLibrary2)
```

Make sure to prefix accordingly the global group symbols that are used in your applications:

```
extern unsigned char YOURGROUPNAME_SHAVE_GROUPgrpdyndata[];
```

8.7.4.2 Starting the new group

The current version of Dynamic Loading infrastructure performs all necessary actions to start only the default group.

If user creates another group, he has to start it manually. The existing master entry point `SHAVE_DEFAULT_GROUP___AllTimeEntryPoint` cannot be used, as it belongs to the default group.

The new master entry point will be named `YOURGROUP___AllTimeEntryPoint`, but current version only handles `SHAVE_DEFAULT_GROUP___AllTimeEntryPoint`.

However, the user is still able to run the shave application by setting manually the value of i0 register with the address of the desired application entry point. Because i0 register is used to keep the next entry point. Afterward, the classic swcStartShave function can be called, having as parameter the master entry point: `SwcStartShave(shave_nr, YOURGROUP___AllTimeEntryPoint)`. This will ensure the execution of the shave application entry.

8.7.5 Module data configuration

Module data represents data which is meant to be overwritten or accessed dynamically at runtime. This data is defined in file `theSubModule.c` and includes the following configurations:

- application entry point
- dynamic data section
- heap memory address
- heap size in bytes
- stack size in bytes

NOTE: Heap memory is initialized by default before being accessed by applications.

This is the data type for module data available in `theDynContext.h`:

```
typedef struct DynamicContext_elm{
    _ExecutionContext_t crtContextInfo[TOTAL_NUM_SHAVES];
    _TorFn_t* ctors_start;
    _TorFn_t* ctors_end;
    _TorFn_t* dtors_start;
    _TorFn_t* dtors_end;
    uint32_t  heap_size;
    uint32_t  stack_size;
    unsigned char* entryPoint;
    DynamicContextInstancesPtr instancesData;
    uint32_t groupEntryPoint;
    unsigned char* appdyndata;
    unsigned char* groupappdyndata;
    unsigned int groupappdyndatasize;
    DYNCONTEXT_HEAP_ACTION_TYPE initHeap;
    DYNCONTEXT_APP_REENTRANT_TYPE reentrant;
    unsigned int cmxCriticalCodeSize;
} DynamicContext_t;
```

- crtContexInfo → keeps track of the execution context for each shave running this app.

- ctors_start, ctors_end → gets initialized with the application specific constructor array information.
- dtors_start, dtors_end → gets initialized with the application specific destructor array information.
- entryPoint → the main entry point for the applications.
- instancesData → this contains the group and heap memory for each instance as well as the number of application instances used and their allocated shaves. This data is handled internally.
- groupEntryPoint → contains the group entry point.
- appdyndata → this contains an address value where the application data required sections are found in DDR. This will be loaded in CMX at runtime.
- groupappdyndata → this is an address value for the group's data requirements shvdlib.
- groupappdyndatasize → this contains the size of the required library group data.
- initHeap → flag informing if the heap should be reinitialized or not at start of SHAVE application.
- reentrant → flag informing if an application is reentrant or not.
- heap_size → contains the size of the desired heapSize.
- stack_size → contains the size of the stack.
- cmxCriticalCodeSize → this field keeps track of desired window A address to be set.

When building your application with Mixed Instantiation infrastructre, each shave application will get its own instance of module data, meaning each will have individual values generated.

All these configurations are accessible via an external global variable in form of

```
extern DynamicContext_t MODULE_DATA(MesgOne);
```

This represents one instance of module data, with all default values generated at build time. In the example above, the module data configurations for `MesgOne` application.

Therefore, all `DynamicContext_t` members are already accessible and can be used or changed in user's application, at runtime.

The build time configurations of module data can be found in *generalsettings.mk* file. Here are defined the default values, if no specific value is set by the user.

```
define cDynContextDefs_template
$(1)_HEAPSIZE       ?= 1024*6
$(1)_STACKSIZE      ?= 1024*6
$(1)_MainEntry      ?= Entry
$(1)_HEAP_INIT      ?= DYNCONTEXT_HEAP_INIT
…
endef
```

The template above is called for each application, so a specific configuration set looks like:

`MesgOne_HEAPSIZE` – default heap size for MesgOne application – 6 kbytes

`MesgOne_STACKSIZE` – default stack size for MesgOne application – 6 kbytes

`MesgOne_MainEntry` – default shave entry point name

`MesgOne_HEAPSIZE` – default heap action – needs initialization

8.7.5.1 Example module data configuration

The following code example shows how the module data configuration can be modified in a user application.

In your Leon OS or Leon RT file, first declare the external module data that needs to be overwritten. Here it is shown for shave application `MesgThree`.

```
extern DynamicContext_t MODULE_DATA(MesgThree);
```

Global variable `MODULE_DATA(MesgThree)` will be expanded by preprocessing to `MesgThree_ModuleData`.

Then the module configuration can be overwritten in the following way, before loading and executing `MesgThree` dynamic library on the shave:

```
void POSIX_Init (void *args)
{
      /* overwrite heap memory size */
      MODULE_DATA(MesgThree).heap_size = 5*1024;
      /* overwrite stack memory size */
      MODULE_DATA(MesgThree).stack_size = 3*1024;
      /* now launch MesgThree application on shave */
      ...
}
```

8.8 Runtime instantiated applications

8.8.1 Overview

This chapter aims to provide user understanding of how runtime instantiated code to SHAVE cores for execution is possible.

8.8.2 File types that are Runtime instantiated

Executable files that have the extension .shvdlib can be runtime instantiated.

These files are created out of “elf” build files. Runtime instantiating is supported through the use of the `OsDrvSvu.h` library functions:

```
void OsDrvSvuLoadShvdlib(u8 *sAddr, u32 targetS)
```

8.8.2.1 Shvdlib file type

`Shvdlib` files are elf object files stripped of any symbols. In order to exemplify their usage, we will take as a study material the example `simpleRTEMS_shaveDynamic`. There are two entry points: `addshave_shaveentry` and `subshave_shaveentry`, as there are two SHAVE applications, where `addshave` and `subshave` represent the `appName`, while `shaveentry` represents the name of the entry point -`shaveentry` existing in both `SHAVE` applications.

The first thing to note is adding the `shvdlib` objects into the application's Makefile. Two raw data files are added: one made out of the sparc-elf-object file and one made out of the sparc-elf-object symbols that hold symbolic information about `shvdlib`:

```
RAWDATAOBJECTFILES  = $(AddApp) _sym.o $(AddApp) _bin.o
RAWDATAOBJECTFILES + = $(SubApp) _sym.o $(SubApp) _bin.o
```

Next, in the SHAVE local build rules, creating a binary file packing the `shvdlib` file:

```
#This creates a binary file packing the shvdlib file
$(AddApp) _bin.o : $(AddApp) .shvdlib
       $(ECHO) $(OBJCOPY)  -I binary --rename-section . data = $(DDR_DATA) \
      --redefine-sym  _binary_ $(subst /,_,$(subst .,_,$<)) _start =adddyn \
      -O elf32-sparc -B sparc $< $@

#App related build options
#This creates a binary file packing the shvdlib file
$(SubApp) _bin.o : $(SubApp) .shvdlib
       $(ECHO) $(OBJCOPY)  -I binary --rename-section . data = $(DDR_DATA) \
      --redefine-sym  _binary_ $(subst /,_,$(subst .,_,$<)) _start =subdyn \
      -O elf32-sparc -B sparc $< $@
```

This rule creates a binary file and stores it into a memory section (DDR in this example). It also provides an `adddyn` or `subdyn` symbol so that we can pass that as parameter to the LEON function that loads `shvdlib` into SHAVEs.

It is possible to enable the adding of the application name context information at the beginning of each `shvdlib`.

This information can be used during debugging to identify the location of the code section for each running shave application.

In order to enable this information, the build system variable `DEBUGLOADCONTEXT` needs to be set to the value `YES`. By default, this variable is set to `NO` from the build system and can be overwritten in the application Makefile.

The application library name information has the following format:

- First 4 bytes will contain length of the name,
- followed by the name of the application library, excluding the file extension that the code and data segment belongs to.
- The actual ELF (shvdlib) content aligned at a memory address which is aligned to a multiple of 4 bytes.

  During runtime, before starting the shave processor for a registered shave application, the name of the application and application group data library will be stored in the following global context variable: `__ShvdlibContextLoaded[shave#]` (it will store the name of the application library loaded).

In order to see the names of the applications loaded, at any given particular moment and on any shaves, the following command should be run in debug mode: for { `set i 0} { $i < 12 } {incr i } { puts [format "%02d %s" $i [mget -asciiz __ShvdlibContextLoaded\[$i\]]] }`.

Then, in the LEON code of the application:

```
//runtime instantiated code

// external variable linked to the shave's entrance address by the gnu
linker script
extern u32 addshave_shaveentry;
extern u32 subshave_shaveentry;

// external var. linked to the address where the dynamically linkable
code segment is stored
extern u8 adddyn;
extern u8 subdyn;

// external var. linked to the address relative to the dynamic library
where these variables are stored
extern u32 addshave_myint1;
u32  shaveNumber = 0;
 // open shave
 osDrvSvuHandler_t chandler;
 osRetCod += OsDrvSvuOpenShave(&chandler, shaveNumber,
OS_MYR_PROTECTION_SELF);

 // Only continue if the shave resource is available
 if (osRetCod != OS_MYR_DRV_SUCCESS)
 {
    printf ("Open shave failed with error code: %d\n",osRetCod);
 }
 else
 {
 // record if any operation failed in osRetCode for later evaluation
    osRetCod += OsDrvSvuSetShaveWindowsToDefault(&chandler);
    osRetCod += OsDrvSvuSetAbsoluteDefaultStack(&chandler);
    osRetCod += OsDrvSvuLoadShvdlib((u8 *)adddyn, &chandler);
    osRetCod += OsDrvSvuResetShave(&chandler);
 // Set input parameters Shaves, note that the input parameters are
now window relative addresses
    osRetCod += OsDrvSvuSolveShaveRelAddr(addshave_myint1,
shaveNumber, &aux_address);
```

```
// Access the data stored on that address:
   aux = (u32 *)aux_address;
   *(aux)   = 4;

//starting shave with the entry point specified at address
   osRetCod += OsDrvSvuRunShave(&chandler, addshave_shaveentry);
}
```

8.9 DDR configuration

Fundamentally there are two scenarios for how DDR can be initialized and configured:

- DDR sections exist in the application's elf or `mvcmd` file.
- No DDR sections.

8.9.1 Application has DDR sections

It is a typical use-case when application has code and/or data placed in DDR memory sections which get built into the application's elf or `mvcmd` file. In this case `moviDebug` or the bootloader recognizes the DDR sections and before loading them it performs the DDR initialization with the following settings:

- PLL1 is supplied from REFCLK0.
- PLL1 is configured to 264 MHz (MA215x) / 456 MHz (MA245x).
- DDR is supplied directly from PLL1 bypassing the divider.
- DDR frequency is 528 MHz (MA215x) / 732 MHz (MA245xB) / 912 MHz (MA245xC).
- DDR is clocked on edges, so the data rate is 32 bit @ 528 MHz (MA215x) / 732 MHz (MA245xB) / 912 MHz (MA245xC).

Since the on-the-fly re-initialization of the DDR is not supported it is important that the application do not touch the DDR settings mentioned above.

If the application runs on RTEMS, than the RTEMS startup code takes care of the PLL1, but it is important for the application not to set the DDR auxillary clock setting. In case the startup clock configuration has to be changed e.g. by calling the `OsDrvCprSetupClocks(pSocClockConfig)` function, then the PLL1 settings should not be changed.

If the application runs on bare-metal, than the CPR configuration has to be done carefully making sure that the DDR clock is not modified in the `pAuxClkCfg` field of the `tySocClockConfig` structure and the `targetPll1FreqKHz` field is set to zero ensuring the PLL1 frequency is not changed. See an example clock configuration for MA215x below:

```
// auxiliary clock settings
tyAuxClkDividerCfg auxClk[] =
    {
        {AUX_CLK_MASK_UART, CLK_SRC_REFCLK0, 96, 625},   // Give the
UART an SCLK that allows it to generate an output baud rate of of
115200 Hz (the uart divides by 16)
        {0,0,0,0}, // Null Terminated List
    };


// system clock settings
tySocClockConfig appClockConfig =
    {
        .refClk0InputKHz        = 12000,           // Default 12MHz
input clock
        .refClk1InputKHz        = 0,               // Assume no
secondary oscillator for now
        .targetPll0FreqKHz      = 480000,
        .targetPll1FreqKHz      = 0,               // set in DDR
driver
        .clkSrcPll1             = CLK_SRC_REFCLK0, // Supply both
PLLS from REFCLK0
```

```
        .masterClkDivNumerator   = 1,
        .masterClkDivDenominator = 1,
        .cssDssClockEnableMask   = DEFAULT_CORE_CSS_DSS_CLOCKS,
        .mssClockEnableMask      = 0,                                  //
Not enabling any MSS clocks for now
        .upaClockEnableMask      = 0,                                  //
Not enabling any UPA clocks for now
        .pAuxClkCfg              = auxClk,
    };
```

8.9.2 No DDR section

If the application elf or `mvcmd` does not contain any DDR section than the DDR will not be initialized by `moviDebug` or bootloader. However applications may want to use the DDR, so in this case the application has to explicitly configure the DDR clocks and then call the DDR initialization function.

Example DDR configuration for MA215x:

```
// auxiliary clock settings
tyAuxClkDividerCfg auxClk[] =
    {
        {AUX_CLK_MASK_DDR_CORE_CTRL, CLK_SRC_PLL1, 1, 1},   // Ensure
DDR always has a clock ,Supply DDR from PLL1 , Run DDR at directly
from PLL (Bypass divider)
        {AUX_CLK_MASK_UART, CLK_SRC_REFCLK0, 96, 625},   // Give the
UART an SCLK that allows it to generate an output baud rate of of
115200 Hz (the uart divides by 16)
        {0,0,0,0}, // Null Terminated List
    };


// system clock settings
tySocClockConfig appClockConfig =
    {
        .refClk0InputKHz         = 12000,           // Default 12MHz
input clock
        .refClk1InputKHz         = 0,               // Assume no
secondary oscillator for now
        .targetPll0FreqKHz       = 480000,
        .targetPll1FreqKHz       = 264000,          // set in DDR
driver
        .clkSrcPll1              = CLK_SRC_REFCLK0, // Supply both
PLLS from REFCLK0
        .masterClkDivNumerator   = 1,
        .masterClkDivDenominator = 1,
        .cssDssClockEnableMask   = DEFAULT_CORE_CSS_DSS_CLOCKS,
        .mssClockEnableMask      = 0,                                  //
Not enabling any MSS clocks for now
        .upaClockEnableMask      = 0,                                  //
Not enabling any UPA clocks for now
        .pAuxClkCfg              = auxClk,
    };


// clock and memory initialization
{
    ...
```

```
    DrvCprInit();
    DrvCprSetupClocks(&appClockConfig);
    ...
    // Timer driver is used by the DDR driver
    DrvTimerInit();
    ...
    DrvDdrInitialise(NULL);
    ...
}
```

8.10 System Resource Management

There are various hardware and software resources in the Myriad 2 system which are broadly used by drivers, components as well as applications. However these resources are limited, so it is important to share them properly to avoid conflict within the software elements of an application.

This chapter lists these system resources and the way they are used in MDK, so that, applications can define their configuration safely.

8.10.1 Hardware Mutexes

Myriad 2 provides 32 hardware mutexes.

Please refer to the following file regarding which hardware mutexes are used by MDK and which are free for application usage:

```
mdk/common/shared/include/swcMutexUsage.h
```

8.10.2 CMX DMA Agents

The CMX DMA driver allows users to configure the CMX DMA Link Agent assignment. At the same time there is also a default configuration which is as follows:

- SIPP Framework uses `DMA_AGENT0` exclusively (cannot be reconfigured).
- CMX DMA driver on Leon OS uses `DMA_AGENT1` and `DMA_AGENT2`.
- CMX DMA driver on Leon RT uses `DMA_AGENT3`.
- CMX DMA driver on SHAVE_X uses `DMA_AGENT[(1+number_of_shave)%3]`.

The access to the same Agent from multiple processors is safe as it is protected by hardware mutexes.

8.10.3 RTEMS events

The RTEMS Classic API provides two sets of events for application and system usage. The RTEMS Event Set provides 32 general purpose events that are purposed to be used by user level applications. The RTEMS System Event Set provides another 32 events purposed to be used by system level code.

Out of the above sets of events, they are used by MDK and RTEMS, as follows:

Event Range	Used for
Events	
`RTEMS_EVENT_0 - RTEMS_EVENT_11`	Available for applications
`RTEMS_EVENT_12 - RTEMS_EVENT_22`	Used by the MDK drivers/components
`RTEMS_EVENT_23 - RTEMS_EVENT_31`	Available for applications
System Events	
`RTEMS_EVENT_0 - RTEMS_EVENT_23`	Used by the MDK drivers/components
`RTEMS_EVENT_24 - RTEMS_EVENT_31`	Available for applications

For the detailed event usage in MDK please refer to the following file:

```
common/drivers/myriad2/socDrivers/leon/rtems/include/OsCommon.h
```

9 Drivers

9.1 Scope of this Section

This section aims at providing basic overview on the usage of drivers. This section will cover typical use cases for drivers provided by the MDK and RTEMS.

9.2 RTEMS Drivers

This section will focus on RTEMS drivers including their integration into the OS.

9.2.1 SD Driver

This section is dedicated to describing how to mount a file system on an SD card, how to configure RTEMS properly, and performance metrics.

9.2.1.1 Overview

The SD Driver provided in RTEMS is a IO Device Driver designed to work with Libblock in order to interface with the file systems provided by the OS. This driver uses the block device API and a software cache, which is built on top of this API.

The root file system is always IMFS (In Memory File System). Our typical use case includes DOS file system with short names. The rest of this chapter is dedicated to explain the usage of this driver and its integration.

9.2.1.2 SD Driver Usage

In order to mount a file system on the SD card we must first register the SD Driver by calling OsDrvSdioInit(osDrvSdioEntries_t *) with custom parameters prior to any operation on the file system.

```
// Struct that keeps the card slot that would be used. The device
name, the maximum speed required and the callback function used to
switch the SD card input voltage.

typedef struct {
   u8 slot;
   const char *devName;
       tyDrvSdioSpeedMode maxSpeed;
       DrvSdioVoltSwitchFunc volSwCallback;
} osDrvSdioEntry_t;

// Struct that keeps the number of card slots; Interrupt Priority that
the would be set for the driver and a slot information for every slot

typedef struct
{
   u8 slots;
   u8 interruptPriority;
   osDrvSdioEntry_t slotInfo[OSDRVSDIO_MAX_SLOTS];
} osDrvSdioEntries_t;
```

We must include the number of slots available, the priority of the interrupts to be generated, and extra information including the name of the driver to be registered per slot.

Once the driver has been registered, we proceed to register logical partitions by using rtems_bd_part_register_from_disk with the name registered for our driver. For more details about this operation, please go to:

https://docs.rtems.org/doxygen/branches/master/group__rtems__bdpart.html

Next step is to actually mount the file system to associate each partition with a mount point. For this operation we need a Mount Table to pass as a parameter to rtems_fsmount such as:

```
static const rtems_fstab_entry fs_table [] = {
   {
     .source = "/dev/sdc0",
     .target = "/mnt/sdcard",
     .type = "dosfs",
     .options = RTEMS_FILESYSTEM_READ_WRITE,
     .report_reasons = RTEMS_FSTAB_NONE,
     .abort_reasons = RTEMS_FSTAB_OK
    },
   {
     .source = "/dev/sdc01",
     .target = "/mnt/sdcard",
     .type = "dosfs",
     .options = RTEMS_FILESYSTEM_READ_WRITE,
     .report_reasons = RTEMS_FSTAB_NONE,
     .abort_reasons = RTEMS_FSTAB_NONE
    }
};
```

This structure allows to mount logical partitions on different mount points, and describes the type of file system to be mounted. For further details about mounting a file system please have a look at:

https://docs.rtems.org/doxygen/branches/master/group__rtems__fstab.html

From the application point of view the user must configure the system to use Libblock and the cache. An example of how to configure the system can be found at:

https://devel.rtems.org/wiki/TBR/UserManual/Using_the_RTEMS_DOS_File_System

As the driver is interfacing an SD card, some considerations must be taken into account to help reduce the overhead introduced by the file system itself. RTEMS will issue a sequence of operations (read/write) to be applied on the SD card. The size of the operations will be limited by the file system itself in the case of DOS. The number of operations issued per transaction is dependent on the cache configuration. So let's say we use clusters of 16K bytes and the following configuration:

```
#define CONFIGURE_BDBUF_MAX_READ_AHEAD_BLOCKS      (16)
#define CONFIGURE_BDBUF_MAX_WRITE_BLOCKS           (64)
#define CONFIGURE_BDBUF_BUFFER_MIN_SIZE            (512)
#define CONFIGURE_BDBUF_BUFFER_MAX_SIZE            (32 * 1024)
#define CONFIGURE_BDBUF_CACHE_MEMORY_SIZE    (4 * 1024 * 1024)
```

With this configuration the system will be able to read 16*16K bytes and write 64*16K bytes in a single transaction, whereas if we had used clusters of 32K, the system would be able to read 16*32K and write 64*32K bytes in a single transaction.

Please note that even if transactions are big enough, the file system may need to synchronize before writing. This means that depending on the state of the file system, it may need to issue some read operations before writing or write to different locations to update file system tables, which may impact the overall write speed significantly. This impact is also dependent on the SD card used, the faster it is, the less this behavior affects the performance.

As part of the MDK, Movidius ships a practical example on how to use this driver: `\mdk\examples\HowTo\rtems_apps\simpleRTEMS_sdCard`.

9.2.1.3 SD Driver Performance

For reference, Movidius has characterized the performance of different SD cards using FAT32 and 32 Kbyte clusters on both MV0182 Rev5 and MV0212 Rev1 running at 600 MHz. For the purpose of this test, the SD cards were configured to work in UHS mode SDR50. The data was obtained by writing 2GB of data in chunks of 20MB bytes and reading them back for verification.

SD Card	MA2150		MA2450	
	Average Write MB/s	Average Read MB/s	Average Write MB/s	Average Read MB/s
SDHC I (3) ExtremePro SanDisk, class 10, 32GB	24.809	24.652	24.729	24.663
SDHC I Sandisk U3, class 10, 16GB	17.238	24.423	17.215	24.743
SDHC I Sony, class 10, U1, 8GB	9.523	24.654	9.187	24.842
micro SDHC I SanDisk, class 10, 32GB	13.240	24.315	13.561	24.588
micro SDHC I SanDisk, class 10, 8GB	4.581	24.192	4.545	24.349
micro SDHC Maxwell, class 10, 8GB	8.754	24.512	8.732	13.148

Note that depending on the file system and the SD card used, write performance may drop occasionally by 10%-50%.

9.2.1.4 SD Driver Limitations

Movidius is working to widen the SD cards supported and fully support partitioned SD cards. Depending on the SD card used, partitioned SD cards may not work properly. For these cases we may use the card without a partition table. Some Windows formatting applications do this by default, such as SDFormatter. Under Linux the procedure would be:

1. Step one: erase the partition table:

```
sudo parted /dev/[your_sdcard_device] mklabel loop
```

2. Step two: format the sdcard:

```
sudo mkdosfs -s 64 -S 512 -I /dev/[your_sdcard_device]
```

A lesser known fact, but one important to note is that for maximum SDCard performance, SDCards should be formatted with the proper amount of reserved sectors. More may be read about this on the internet on links such as, for example:

http://3gfp.com/2014/07/formatting-sd-cards-for-speed-and-lifetime/

Although it is not mandatory to take these tips into account the speed improvements may be quiet noticable. Not formatting cards in this way may result in speed losses of up to 60%. For convenience, this is a small script (formatProper.sh) to do this for 16 GB cards:

```
////////////////////////////////////////////
#!/bin/bash
DEVICE=$1
#start by deleting the partition table
parted $DEVICE mklabel loop
FATSize=`mkdosfs -n OPTIMSD -s 64 -S 512 -I -v $DEVICE | grep "FAT
size" | cut -d ' ' -f 4`
GOODRESERVED=`expr 8192 \- $FATSize \* 2`
echo For this card $GOODRESERVED sectors will be reserved
#Do the proper formatting
mkdosfs -n OPTIMSD -s 64 -S 512 -R $GOODRESERVED -I -v $DEVICE > /dev/
null
```

To run this script just type:

```
sudo ./formatProper.sh /dev/[your_sdcard_device]
```

9.2.2 Resource Manager

9.2.2.1 Overview

The Resource Manager (ResMgr) is the driver responsible for allocating HW resources to the different CPUs, ensuring that several different CPUs do not use the same HW block at the same time.

The idea is that, before accessing any HW resource, any SW component should have this resource allocated by the ResMgr. If the allocation is granted, the SW component can use it at will, with the guarantee that no other CPU will interfere with it. Once the SW component is done using a given resource, it can release it so that another CPU can access it if needed. If the allocation is not granted, the SW component must not try to access the resource and should either do something else or wait for the resource to become available.

ResMgr also provides a couple of other features:

- Get the current frequency of some of the Myriad X chip's clocks.
- Execute a function under the protection of a mutex.

9.2.2.2 Resources needed

ResMgr driver allocates one RTEMS semaphore for its internal use.

9.2.2.3 API

In RTEMS, ResMgr is controlled exclusively through ioctl commands.

9.2.2.3.1 Allocation

The basic allocation is done using the **`OS_DRV_RESMGR_ALLOCATE`** command. The parameter for this command is of type **`OsDrvResMgrRequest *`**.

OsDrvResMgrRequest is a structure containing the following fields:

- `uint8_t type`: The type of resource requested. Can be any type defined by ResMgr (see defines `OS_DRV_RESMGR_SHAVE` to `OS_DRV_RESMGR_MUTEX` in `OsDrvResMgr.h`)
- `union alloc_by`: This union has the following sub-fields:
  - `uint8_t id`: This field is the one that should be used for resources of any type except `OS_DRV_RESMGR_MUTEX`. Can be either the numeric ID of a specific resource of a given type (e.g. 0 to 3 for UART0 to UART3) or `OS_DRV_RESMGR_REQ_ID_ANY` if the ID does not matter.
  - `void *info`: This field is the one that should be used for mutexes (`OS_DRV_RESMGR_MUTEX`). This can be any 32-bit numeric value identifying a given mutex. Typically, this would be the address of the data the mutex is used to protect the access to.
- `OsDrvResMgrHandler *handler`: Pointer on the handler on the resource. The handler must be allocated by the requester. If the allocation is successful, the characteristics of the allocated resource are written in the handler so that the requester can check them. The fields in this structure are:
  - `uint8_t type`: The type of resource allocated. Should be equal to the type of resource requested.
  - `uint8_t id`: ID of the resource allocated. Should be equal to the ID requested if specified or any valid resource ID if `OS_DRV_RESMGR_REQ_ID_ANY` was used. In the case of a mutex, the ID will be the ID of the mutex already allocated to protect the same data (same `info` value) or any ID if no mutex is already allocated for this data.
- `int32_t req_status`: Returns the status of the request. Can be:
  - `RTEMS_SUCCESSFUL`: Allocation granted.
  - `RTEMS_INVALID_ADDRESS`: At least one of the parameters has an invalid value.
  - `RTEMS_NOT_OWNER_OF_RESOURCE`: The requester is already the owner of the resource.
  - `RTEMS_RESOURCE_IN_USE`: No resource available to satisfy the request.

Alternatively, several resources can be requested at the same time, using the **OS_DRV_RESMGR_ALLOCATE_GROUP** command. The parameter for this command is of type **OsDrvResMgrAllocateGroupParam ***.

OsDrvResMgrAllocateGroupParam is a structure containing the following fields:

- `OsDrvResMgrRequest *request`: Pointer to the first element of an array of `OsDrvResMgrRequest` elements. There must be one request structure by resource to allocate. All combinations of type, IDs/info, etc. are allowed but some will obviously fail (e.g. if trying to allocate twice a resource of the same type and ID). Each allocated resource must have its own handler.
- `uint8_t req_nb`: Number of allocation requests to handle.
- `uint8_t atomic`: If 1, either all resources are granted or none. If 0, each resource will be individually granted or not (status for each allocation can be checked using the `req_status` field in the corresponding request structure).

9.2.2.3.2 Release

The basic release is done using the **OS_DRV_RESMGR_RELEASE** command. The parameter for this command is of type **OsDrvResMgrHandler ***. It must be the handler passed to ResMgr on allocation of the resource the user wants to release (see section on Allocation for more details).

Alternatively, several resources can be released at the same time, using the **OS_DRV_RESMGR_RELEASE_GROUP** command. The parameter for this command is of type **OsDrvResMgrReleaseGroupParam ***.

OsDrvResMgrReleaseGroupParam is a structure containing the following fields:

- `OsDrvResMgrHandler *handler`: Pointer to the first element of an array of OsDrvResMgrHandler elements. There must be one handler by resource to release.
- `uint8_t res_nb`: Number of resources to release.
- `uint8_t release_nb`: Returns the number of resources actually released. If the process is successful, this will be equal to "res_nb". If one of the release fails, the function will stop directly without trying to release the next ones (in the order of the array of handlers).

9.2.2.3.3 Clock values

Some of the clocks in the system can be dynamically configured and knowing their frequency can be needed by some SW components. To handle that, ResMgr stores, at any time, the current values of those clocks' frequencies.

The user can request the frequencies of all these clocks by using the **OS_DRV_RESMGR_GET_CLOCK_ENTRY** command. The parameter for this command is of type **OsDrvResMgrInfoClockEntry ***.

OsDrvResMgrInfoClockEntry is a structure containing the following fields:

- `uint32_t osc1`: Returns the frequency of oscillator 1 (in kilohertz).
- `uint32_t osc2`: Returns the frequency of oscillator 2 (in kilohertz).
- `uint32_t pll1_freq_KHz`: Returns the frequency of PLL 1 (in kilohertz).
- `uint32_t pll2_freq_KHz`: Returns the frequency of PLL 2 (in kilohertz).
- `uint32_t system_freq_KHz`: Returns the frequency of system clock (in kilohertz).

Alternatively, the user can request only the frequency of a given clock by using the **OS_DRV_RESMGR_GET_CLOCK_ENTRY_FIELD** command. The parameter for this command is of type **OsDrvResMgrClockEntryFieldParam ***.

OsDrvResMgrClockEntryFieldParam is a structure containing the following fields:

- `OsDrvResMgrClk clock_entry`: `OsDrvResMgrClk` is an enumerate used to request which clock the user wants the frequency of. Values can be:
  - `OS_DRV_RESMGR_OSC1`
  - `OS_DRV_RESMGR_OSC2`
  - `OS_DRV_RESMGR_PLL1`
  - `OS_DRV_RESMGR_PLL2`
  - `OS_DRV_RESMGR_SYSCLK`
- `uint32_t value`: Returns the frequency of the requested clock.

9.2.2.3.4 Protected execution

To avoid interference of other CPUs when executing a given function, ResMgr provides the possibility of executing this function under the protection of a HW mutex.

This can be done using the **OS_DRV_RESMGR_EXECUTE_PROTECTED** command. The parameter for this command is of type **OsDrvResMgrExecuteProtectedParam ***.

OsDrvResMgrExecuteProtectedParam is a structure containing the following fields:

- `int32_t (*protected_func)(void *)`: The function to execute. The prototype of the function must follow the one defined here.
- `void *arg`: The argument passed to the function to execute.

- `int32_t res`: Returns the value returned by the function after execution.
- `int32_t level`: Level of protection. Can be 0 or 1. The function to execute is protected only against the execution of other functions using the same level of protection.

9.3 SHAVE Drivers

9.3.1 Resource Manager

9.3.1.1 API

9.3.1.1.1 Allocation

The basic allocation is done using the **`ShDrvResMgrAllocate`** function. The parameters for this function are:

- **`ShDrvResMgrRequest *request`**: This is a structure containing the following fields:
  - `uint8_t type`: The type of resource requested. Can be any type defined by ResMgr (see defines SH_DRV_RESMGR_SHAVE to SH_DRV_RESMGR_MUTEX in ShDrvResMgr.h)
  - `union alloc_by`: This union has the following sub-fields:
    - `uint8_t id`: This field is the one that should be used for resources of any type except SH_DRV_RESMGR_MUTEX. Can be either the numeric ID of a specific resource of a given type (e.g. 0 to 3 for UART0 to UART3) or SH_DRV_RESMGR_REQ_ID_ANY if the ID does not matter.
    - `void *info`: This field is the one that should be used for mutexes (SH_DRV_RESMGR_MUTEX). This can be any 32-bit numeric value identifying a given mutex. Typically, this would be the address of the data the mutex is used to protect the access to.
  - `ShDrvResMgrHandler *handler`: Pointer on the handler on the resource. The handler must be allocated by the requester. If the allocation is successful, the characteristics of the allocated resource are written in the handler so that the requester can check them. The fields in this structure are:
    - `uint8_t type`: The type of resource allocated. Should be equal to the type of resource requested.
    - `uint8_t id`: ID of the resource allocated. Should be equal to the ID requested if specified or any valid resource ID if SH_DRV_RESMGR_REQ_ID_ANY was used. In the case of a mutex, the ID will be the ID of the mutex already allocated to protect the same data (same `info` value) or any ID if no mutex is already allocated for this data.
  - `int32_t req_status`: Returns the status of the request. It can be:
    - `MYR_DRV_SUCCESS`: Allocation granted.
    - `MYR_DRV_INVALID_PARAM`: At least one of the parameters has an invalid value.
    - `MYR_DRV_OWNER_ERR`: The requester is already the owner of the resource.
    - `MYR_DRV_RESOURCE_BUSY`: No resource available to satisfy the request.

This function returns the same value than the one recorded in `req_status` field.

Alternatively, several resources can be requested at the same time, using the **`ShDrvResMgrAllocateGroup`** function. The parameters for this function are:

- **`ShDrvResMgrRequest request[]`**: Array of `ShDrvResMgrRequest` elements. There must be one request structure by resource to allocate. All combinations of type, IDs/info, etc. are

allowed but some will obviously fail (e.g. if trying to allocate twice a resource of the same type and ID). Each allocated resource must have its own handler.

- **`uint8_t req_nb`**: Number of allocation requests to handle.
- **`uint8_t atomic`**: If 1, either all resources are granted or none. If 0, each resource will be individually granted or not (status for each allocation can be checked using the "req_status" field in the corresponding request structure).

This functions returns `MYR_DRV_SUCCESS` or `MYR_DRV_INVALID_PARAM` depending on the case if not in the atomic case. If atomic is set, the function returns the reason of failure of the first non-granted allocation request (see `req_status` values above).

9.3.1.1.2 Release

The basic release is done using the **`ShDrvResMgrRelease`** function. The parameters for this function are:

- **`ShDrvResMgrHandler *handler`**: It must be the handler passed to ResMgr on allocation of the resource the user wants to release (see the section on Allocation for more details).

This function returns the following values:

- `MYR_DRV_SUCCESS`: Release successful.
- `MYR_DRV_INVALID_PARAM`: At least one of the parameters has an invalid value.
- `MYR_DRV_OWNER_ERR`: The caller is not the owner of the resource.

Alternatively, several resources can be released at the same time, using the **`ShDrvResMgrReleaseGroup`** function. The parameters for this function are:

- **`ShDrvResMgrHandler handler[]`**: Array of `ShDrvResMgrHandler` elements. There must be one handler by resource to release.
- **`uint8_t res_nb`**: Number of resources to release.
- **`uint8_t *release_nb`**: Returns the number of resources actually released. If the process is successful, this will be equal to "res_nb". If one of the release fails, the function will stop directly without trying to release the next ones (in the order of the array of handlers).

This function returns the same values than `ShDrvResMgrRelease`. If not `MYR_DRV_SUCCESS`, the error is applicable to the first release which failed.

9.3.1.1.3 Clock values

Some of the clocks in the system can be dynamically configured and knowing their frequency can be needed by some SW components. To handle that, ResMgr stores, at any time, the current values of those clocks' frequencies.

The user can request the frequencies of all these clocks by using the **`ShDrvResMgrGetClockEntry`** function. The parameters for this function are:

- **`ShDrvResMgrInfoClockEntry *infoclk`**: This is a structure containing the following fields:
  - `uint32_t osc1`: Returns the frequency of oscillator 1 (in kilohertz).
  - `uint32_t osc2`: Returns the frequency of oscillator 2 (in kilohertz).
  - `uint32_t pll1_freq_KHz`: Returns the frequency of PLL 1 (in kilohertz).
  - `uint32_t pll2_freq_KHz`: Returns the frequency of PLL 2 (in kilohertz).
  - `uint32_t system_freq_KHz`: Returns the frequency of system clock (in kilohertz).

This functions returns `MYR_DRV_SUCCESS` or `MYR_DRV_INVALID_PARAM` depending on the case.

Alternatively, the user can request only the frequency of a given clock by using the **`ShDrvResMgrGetClockEntryField`** function. The parameters for this function are:

- **ShDrvResMgrClk clock_entry**: ShDrvResMgrClk is an enumerate used to request which clock the user wants the frequency of. Values can be:
  - SH_DRV_RESMGR_OSC1
  - SH_DRV_RESMGR_OSC2
  - SH_DRV_RESMGR_PLL1
  - SH_DRV_RESMGR_PLL2
  - SH_DRV_RESMGR_SYSCLK
- **uint32_t *value**: Returns the frequency of the requested clock.

This functions returns MYR_DRV_SUCCESS or MYR_DRV_INVALID_PARAM depending on the case.

9.3.1.1.4 Protected execution

To avoid interference of other CPUs when executing a given function, ResMgr provides the possibility of executing this function under the protection of a HW mutex.

This can be done using the **ShDrvResMgrExecuteProtected** function. The parameters for this function are:

- **int32_t level**: Level of protection. Can be 0 or 1. The function to execute is protected only against the execution of other functions using the same level of protection.
- **int32_t (*protected_func)(void *)**: The function to execute. The prototype of the function must follow the one defined here.
- **void *arg**: The argument passed to the function to execute.
- **int32_t *res**: Returns the value returned by the function after execution.

This functions returns MYR_DRV_SUCCESS or MYR_DRV_INVALID_PARAM depending on the case.

9.4 Bare Metal Drivers

9.4.1 Resource Manager

9.4.1.1 API

9.4.1.1.1 Allocation

The basic allocation is done using the **DrvResMgrAllocate** function. The parameters for this function are:

- **DrvResMgrRequest *request**: This is a structure containing the following fields:
  - uint8_t type: The type of resource requested. Can be any type defined by ResMgr (see defines DRV_RESMGR_SHAVE to DRV_RESMGR_MUTEX in DrvResMgr.h).
  - union alloc_by: This union has the following sub-fields:
    - uint8_t id: This field is the one that should be used for resources of any type except DRV_RESMGR_MUTEX. Can be either the numeric ID of a specific resource of a given type (e.g. 0 to 3 for UART0 to UART3) or DRV_RESMGR_REQ_ID_ANY if the ID does not matter.
    - void *info: This field is the one that should be used for mutexes (DRV_RESMGR_MUTEX). This can be any 32-bit numeric value identifying a given mutex. Typically, this would be the address of the data the mutex is used to protect the access to.
  - DrvResMgrHandler *handler: Pointer on the handler on the resource. The handler must be allocated by the requester. If the allocation is successful, the characteristics of the allocated resource are written in the handler so that the requester can check them. The fields in this structure are:
    - uint8_t type: The type of resource allocated. Should be equal to the type of resource requested.
    - uint8_t id: ID of the resource allocated. Should be equal to the ID requested if specified or any valid resource ID if DRV_RESMGR_REQ_ID_ANY was used. In the case of a mutex, the ID will be the ID of the mutex already allocated to protect the same data (same info value) or any ID if no mutex is already allocated for this data.
  - int32_t req_status: Returns the status of the request. It can be:
    - MYR_DRV_SUCCESS: Allocation granted.
    - MYR_DRV_INVALID_PARAM: At least one of the parameters has an invalid value.
    - MYR_DRV_OWNER_ERR: The requester is already the owner of the resource.
    - MYR_DRV_RESOURCE_BUSY: No resource available to satisfy the request.

This function returns the same value than the one recorded in req_status field.

Alternatively, several resources can be requested at the same time, using the **DrvResMgrAllocateGroup** function. The parameters for this function are:

- **DrvResMgrRequest request[]**: Array of DrvResMgrRequest elements. There must be one request structure by resource to allocate. All combinations of type, IDs/info, etc. are allowed but some will obviously fail (e.g. if trying to allocate twice a resource of the same type and ID). Each allocated resource must have its own handler.
- **uint8_t req_nb**: Number of allocation requests to handle.
- **uint8_t atomic**: If 1, either all resources are granted or none. If 0, each resource will be individually granted or not (status for each allocation can be checked using the “req_status” field in the corresponding request structure).

This functions returns MYR_DRV_SUCCESS or MYR_DRV_INVALID_PARAM depending on the case if not in the atomic case. If atomic is set, the function returns the reason of failure of the first non-granted allocation request (see req_status values above).

9.4.1.1.2 Release

The basic release is done using the **DrvResMgrRelease** function. The parameters for this function are:

- **DrvResMgrHandler *handler**: It must be the handler passed to ResMgr on allocation of the resource the user wants to release (see the section on Allocation for more details).

This function returns the following values:

- MYR_DRV_SUCCESS: Release successful.
- MYR_DRV_INVALID_PARAM: At least one of the parameters has an invalid value.
- MYR_DRV_OWNER_ERR: The caller is not the owner of the resource.

Alternatively, several resources can be released at the same time, using the **DrvResMgrReleaseGroup** function. The parameters for this function are:

- **DrvResMgrHandler handler[]**: Array of DrvResMgrHandler elements. There must be one handler by resource to release.
- **uint8_t res_nb**: Number of resources to release.
- **uint8_t *release_nb**: Returns the number of resources actually released. If the process is successful, this will be equal to "res_nb". If one of the release fails, the function will stop directly without trying to release the next ones (in the order of the array of handlers).

This function returns the same values than **DrvResMgrRelease**. If not MYR_DRV_SUCCESS, the error is applicable to the first release which failed.

9.4.1.1.3 Clock values

Some of the clocks in the system can be dynamically configured and knowing their frequency can be needed by some SW components. To handle that, ResMgr stores, at any time, the current values of those clocks' frequencies.

The user can request the frequencies of all these clocks by using the **DrvResMgrGetClockEntry** function. The parameters for this function are:

- **DrvResMgrInfoClockEntry *infoclk**: This is a structure containing the following fields:
  - uint32_t osc1: Returns the frequency of oscillator 1 (in kilohertz).
  - uint32_t osc2: Returns the frequency of oscillator 2 (in kilohertz).
  - uint32_t pll1_freq_KHz: Returns the frequency of PLL 1 (in kilohertz).
  - uint32_t pll2_freq_KHz: Returns the frequency of PLL 2 (in kilohertz).
  - uint32_t system_freq_KHz: Returns the frequency of system clock (in kilohertz).

This functions returns MYR_DRV_SUCCESS or MYR_DRV_INVALID_PARAM depending on the case.

Alternatively, the user can request only the frequency of a given clock by using the **DrvResMgrGetClockEntryField** function. The parameters for this function are:

- **DrvResMgrClk clock_entry**: DrvResMgrClk is an enumerate used to request which clock the user wants the frequency of. Values can be:
  - DRV_RESMGR_OSC1
  - DRV_RESMGR_OSC2
  - DRV_RESMGR_PLL1
  - DRV_RESMGR_PLL2
  - DRV_RESMGR_SYSCLK

- **`uint32_t *value`**: Returns the frequency of the requested clock.

This functions returns `MYR_DRV_SUCCESS` or `MYR_DRV_INVALID_PARAM` depending on the case.

9.4.1.1.4 Protected execution

To avoid interference of other CPUs when executing a given function, ResMgr provides the possibility of executing this function under the protection of a HW mutex.

This can be done using the **`DrvResMgrExecuteProtected`** function. The parameters for this function are:

- **`int32_t level`**: Level of protection. Can be 0 or 1. The function to execute is protected only against the execution of other functions using the same level of protection.
- **`int32_t (*protected_func)(void *)`**: The function to execute. The prototype of the function must follow the one defined here.
- **`void *arg`**: The argument passed to the function to execute.
- **`int32_t *res`**: Returns the value returned by the function after execution.

This functions returns `MYR_DRV_SUCCESS` or `MYR_DRV_INVALID_PARAM` depending on the case.

10 Components

The MDK offers to applications ready to use blocks of software which abstract and integrate one or more HW and drivers, in order to implement a specific functionality.

10.1 CamGeneric

10.1.1 Introduction

The CamGeneric component is a layer which abstract the multiple drivers layer for an application which want to use a camera sensor with Myriad 2 chip.

The component support serial MIPI connected cameras and parallel bus connected cameras on up to 24 lines of data. The data flow received by serial MIPI will pass through the Myriad 2 MIPI HW and will be internally handled by a SIPP MIPI Rx receiver or a CIF receiver, depending on the MSS multiplexer configuration. The parallel bus cameras are always handled by a CIF HW block.

An overview of the CamGeneric environment is shown below:

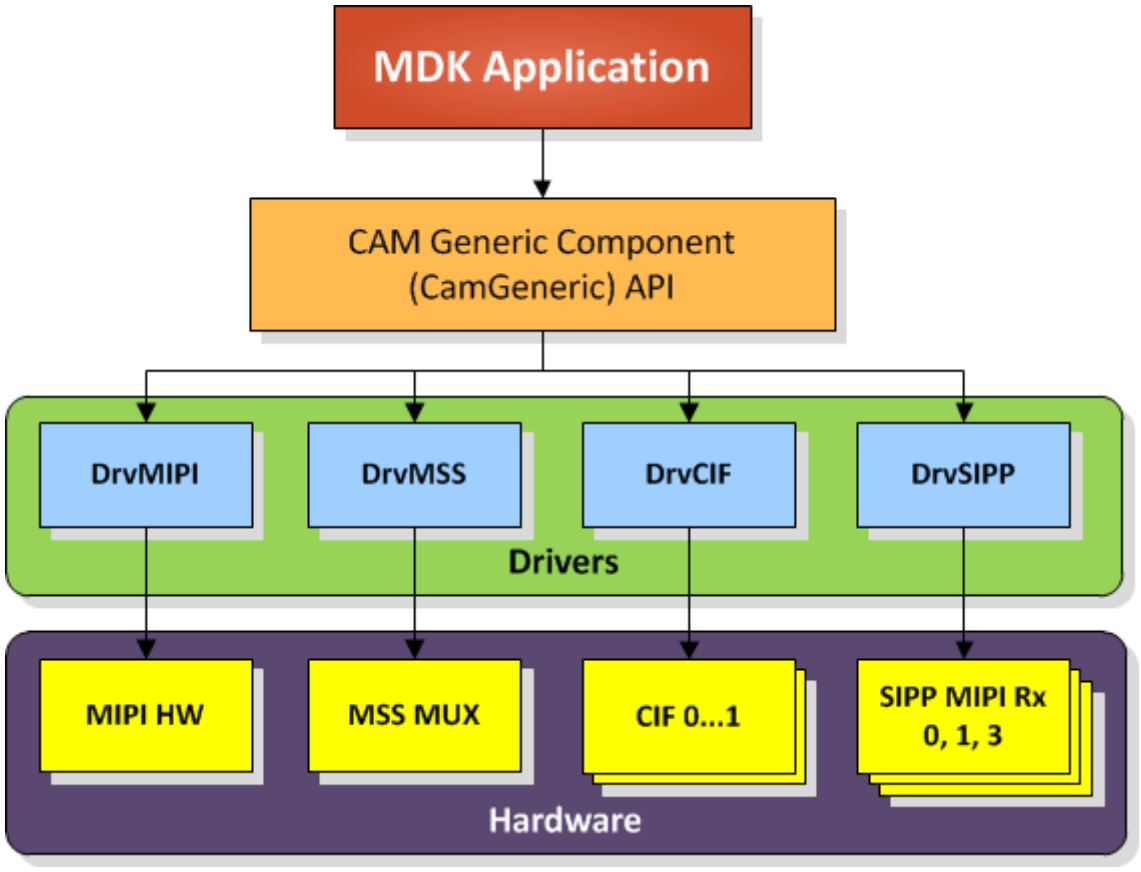


Figure 10: CamGeneric environment overview

More then these main drivers, the CamGeneric also abstract the usage of **DrvGpio** (for parallel bus synchronization signals and sensor pin reset), **DrvIcb** (for the management of the interrupts), **DrvI2C** (for the I2C communication with the sensors).

Beside the abstracted drivers, the CamGeneric depends on helper drivers like DrvTimer (for precise time counting) and DrvCpr (for parallel bus cameras synchronization generation).

The CamGeneric also depend on another component, CameraModules, which contains all sensor related information.

The maximum number of simultaneous instances for CamGeneric managed cameras is 5:

- 2 MIPI/parallel connected cameras on CIF, and
- 3 MIPI cameras on SIPP MIPI Rx.

The CamGeneric component is foreseen to run on Leon OS or Leon RT and can generate interrupts to both of them, meaning to the current running Leon and to the other one (interrupts rerouting).

The CamGeneric must be run on bare metal Leon, no RTOS (RTEMS) integration supported yet.

CamGeneric component is located at **common/components/CamGeneric**. The related sensor configurations can be found in **common/components/CameraModules** and an example demonstrating the usage are located at **examples/Demo.**

10.1.2 Description

The CamGeneric offers to applications an API composed of the following functions:

- `CamInit()`,
- `CamStart()`,
- `CamStop()`,
- `CamStandby()`,
- `CamWakeup()`,
- `CamSetupCallbacks()`,
- `CamSetupInterrupts()` and
- `CamGetFrameCounter()`.

Before using the API, an application must prepare configuration information, then will give this to CamGeneric through the CamInit() function or can give it later, through CamSetupCallbacks(), CamSetupInterrupts(), in the inactive phases of the camera. The user can also bypass the CamGeneric (directly access to HW/drivers), if in need for advanced functionalities not implemented yet, while the camera is inactive (standby).

There are two inactive phases (standby phases) which can be use for camera reconfiguration:

- right after CamInit, before starting the streaming by CamStart (the camera is left in a hot standby state, right after the initialization step).
- after a CamStandby, before CamWakeup (the camera can be in either hot or cold standby).

10.1.2.1 Standby types

For power saving or for time saving reasons and also for reconfigurability reason, the camera can be put in standby. There are two types of standby:

- **HOT STANDBY**: the sensor is deactivated (but still configured), the MIPI controller and PHY and the MSS connections are still active and the CIF/SIPP are reconfigured but not restarted. This standby type is mostly used to save processor time by suspending the interrupts and is fast to recover from (activation last less then 0.5 milliseconds).
- **COLD STANDBY**: same as hot standby plus the sensor is Wakeup out of this state implies full sensor reconfiguration. The wake up duration may last tens of milliseconds, depending on the number of the sensor registers to configure. This standby type saves power.

10.1.2.2 Sensor configuration

The CamGeneric offers two way of configuring the sensor:

- by using some default sensor configuration functions implemented in CamGeneric, for every major action like init/start/standby/wakeup/stop. These are always using I2C communication.
- by calling the custom sensor configuration callbacks that were provided by CamInit or CamSetupCallbacks.

The sensor default functions perform the following actions:

- **default sensor power up** (called by CamInit/CamWakeup): apply a sensor reset by flipping the configured GPIO, then write via I2C in the sensor registers grouped in the first N-2 configuration steps out of sensor header file (see the configuration chapter)
- A customization may be performed even at this default step: the GPIO reset sequence may be skipped if the reset GPIO is set to 0xFF. In this case, the application must perform its own sensor reset sequence before calling the CamInit (no callback provided).
- **default sensor wakeup** (called by CamStart): written via I2C in the sensor registers grouped in the step N-1, usually one start streaming register
- **default sensor standby** (called by CamStandby(`CAM_STATUS_HOT_STANDBY`)): written via I2C in the sensor registers grouped in the last configuration step (step N), usually one stop streaming register
- **default sensor power down** (called by CamStop() and by CamStandby(`CAM_STATUS_COLD_STANDBY`)): written by I2C in the sensor registers grouped in the last configuration step (step N, usually one stop register), then perform sensor reset by GPIO flipping, if GPIO is not 0xFF (otherwise the application must perform its own sensor reset sequence before calling CamStandby(...))

The custom functions will be called instead of the default functions, if the relevant function pointer positions are filled in the callbacksListType structure (offered at CamInit or CamSetupCallbacks).

10.1.2.3 Interrupts management

There are three specific actions the CamGeneric can perform related to the CIF/SIPP interrupts:

- **Buffers management:** the CamGeneric can be configured to initially configure and periodically update (on interrupts) the new buffers addresses in HW, by setting the **getFrame** OR **getBlock** function pointer in the callbacks parameter of CamInit or CamSetupInterrupts**.** The internal ISR will call the user supplied management callback in order to obtain a new buffer address.

  As additional configuration, the application has to provide, by the same API, one event which will trigger the local management in Camgeneric (DMA done, EOF, EOL or other).

  If there is no getFrame nor getBlock function provided (both are NULL), then there is no buffer/HW management in CamGeneric ISR. This means that the applications is responsible to either start/update the ISR infrastructure or implement non ISR related mechanisms (e.g. pooling).
- **HW events notifications:** the CamGeneric will notify the application on any specified HW event, by setting the **notification** function pointer in the callbacks parameter of CamInit or CamSetupInterrupts**.** This mean it will enable the interrupts generation and routing and will handle by it's internal ISR the interrupts events. The internal ISR will only call the user supplied ISR notification callback.

  Additionally to notification pointer setting, the application has to provide, by the same API, one or more events which will trigger the notification by Camgeneric (DMA done, EOF, EOL, errors, ...).

  If there is no notification function provided (and no management function neither), then the CamGeneric will not start the ISRs and the application has to pool for the HW events.
- **Interrupts clearing:** this is always performed when either management or notification are to be done by CamGeneric. But t he application has to provide the list of event s which will be cleared by Camgeneric (DMA done, EOF, EOL and/ or others). Usually, these are the events which got managed and/or notified,

but another content for clearing might be an option (e.g., clearing ALL the current events on CIF, in order that the exceptional – error events get also cleared).

So, by setting the ISR callbacks (in CamInit / CamSetupInterrupts), the application tells CamGeneric **what** to do (management and/or notification or none), and by setting specific contents in management/notification/clearing fields (in CamSetupInterrupts) it tells **on which events** to do.

NOTE: Interrupts specifics for CIF receivers on MA2x5x

1. When one or both CIF receivers are used for connecting a MIPI sensor, the system is set up to generate an additional interrupt from the MIPI controller, as a helper for the CIF windowing logic. This interrupt is not exported to the user through a callback function, but is only used internally by CamGeneric.

   This also means that the **MIPI_IRQ** (= **51** on LRT) and a rerouted isr line (if CamGeneric to run on LOS; configurable through the **ROUTED_IRQ_MIPI**) needs to be reserved for use by CamGeneric.
2. The usage of CIF receivers with parallel connected sensors is not (yet) available on MA2x5x.

10.1.3 Usage scenarios

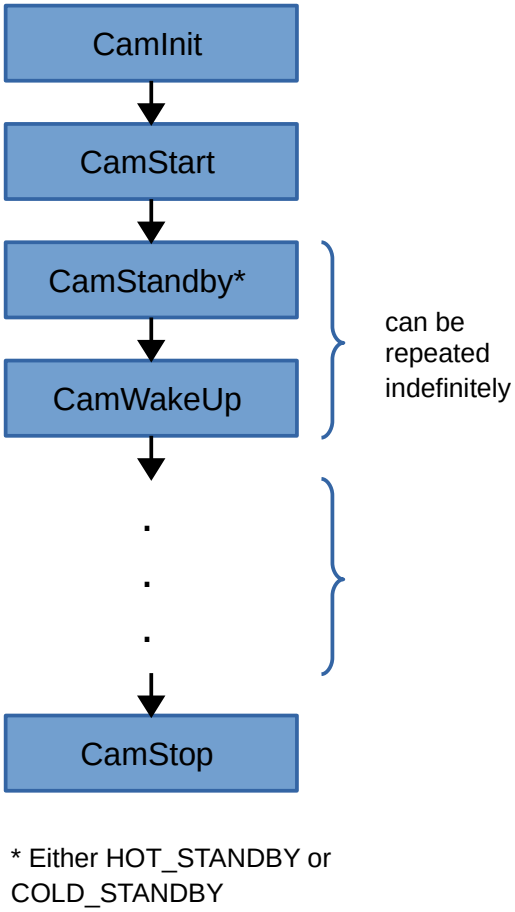


Figure 11: Simple CamGeneric Scenario

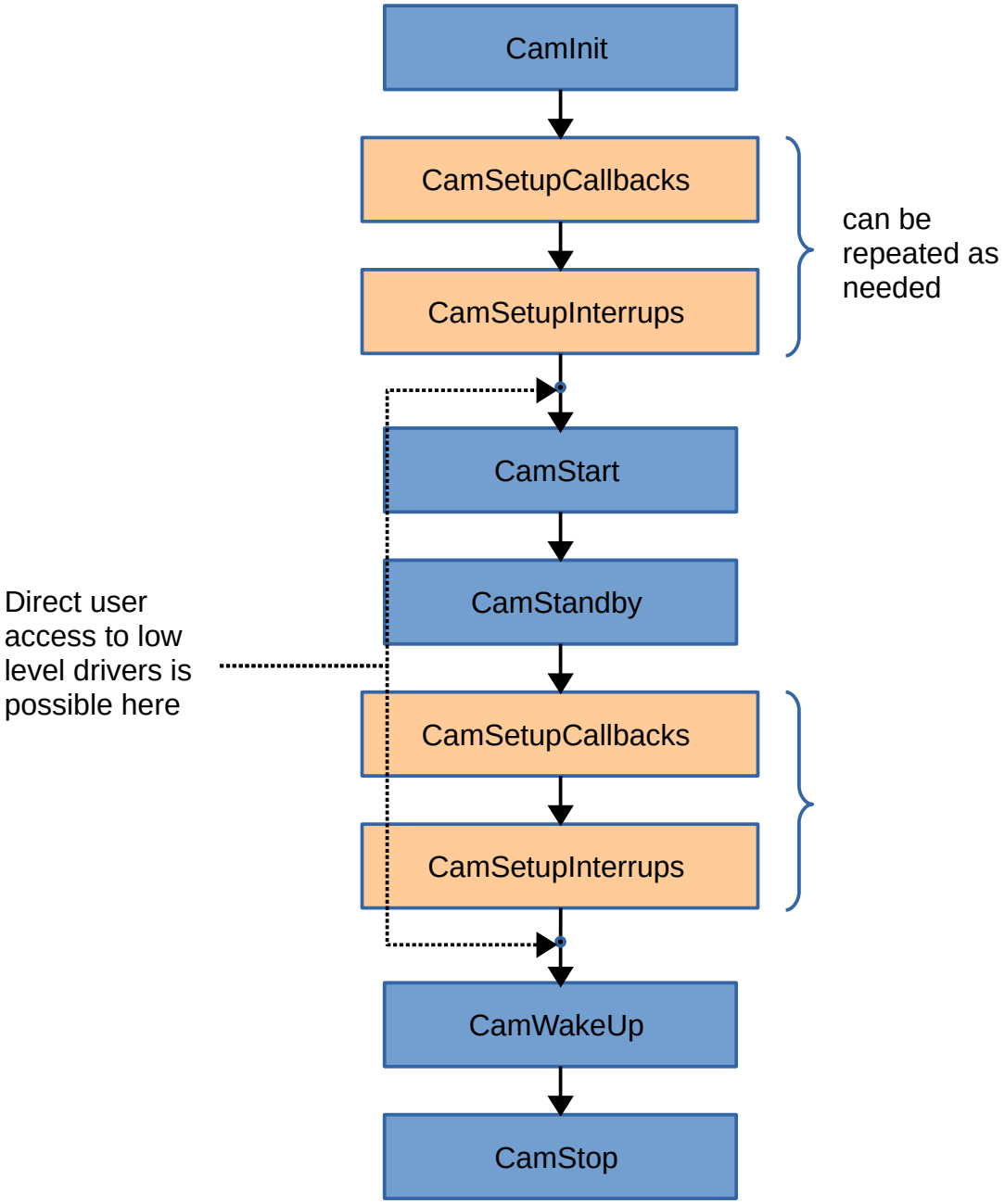


Figure 12: Camgeneric Scenario with low level driver access

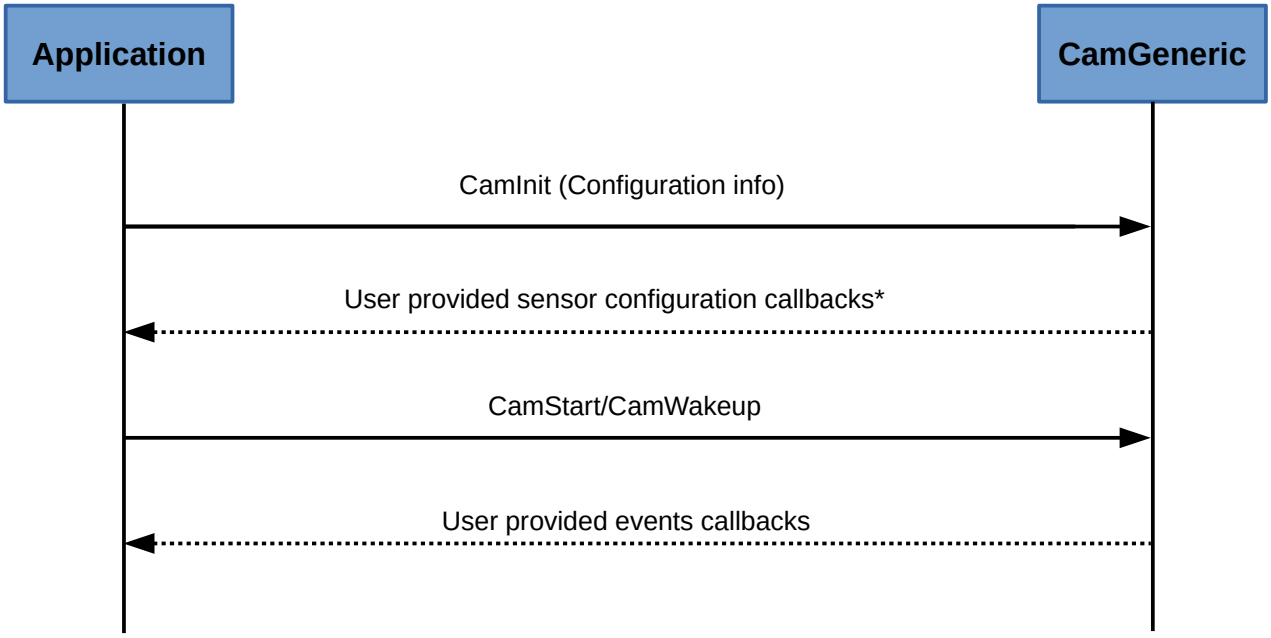


Figure 13: CamGeneric Internals

10.1.4 API

All the pointer parameters sent to CamGeneric point to disposable memory allocated in the application space. The camera handlers is also allocated in the application space and will gather all the needed info for the lifetime of the camera (until stopped).

- **camErrorType CamInit(GenericCameraHandle *hndl, GenericCamSpec *camSpec, CamUserSpec *userSpec, callbacksListStruct* cbList, I2CM_Device *pI2cHandle)**

  This will configure all the needed internal blocks and the sensor and will let the system in a HOT STANDBY state (ready to start). Will return (as parameter) a handler pointer specific for the initialized camera Parameters:

  - hndl: pointer to the camera handle.
  - camSpec: pointer to a sensor configuration structure.
  - userSpec: pointer to a structure of user requirements.
  - cbList: pointer to a structure of callbacks pointers.
  - pI2cHandle: pointer to an I2C handle.

  Return parameter:

  - CAM_SUCCESS if camera was started OK, other values if it failed.

- **camErrorType CamStart(GenericCameraHandle *hndl)**

  This will start the Myriad components and the sensor.

  Parameters:

  - hndl: pointer to the initialized camera handle.

Return parameter:

- CAM_SUCCESS if camera was started OK, other values if it failed.

➢ **camErrorType CamStandby(GenericCameraHandle *hndl, camStatus_type standbyType)**

This will put the sensor and related myriad logic in a standby mode:

Parameters:

- hndl: pointer to the initialized camera handle.
- standbyType: type of the standby: cold or hot standby.

Return parameter:

- CAM_SUCCESS if camera was successfully put in standby, other values if it failed.

➢ **camErrorType CamWakeup(GenericCameraHandle *hndl)**

This will wake up the sensor and related myriad logic from the standby mode

Parameters:

- hndl: pointer to the initialized camera handle
- Return parameter:
- CAM_SUCCESS if camera was successfully waked up, other values if it failed

➢ **camErrorType CamStop (GenericCameraHandle *hndl)**

Resets the sensor and the related myriad logic and clean up the handler

Parameters:

- hndl: pointer to the initialized camera handle

Return parameter:

- CAM_SUCCESS if camera was successfully put in standby, other values if it failed

➢ **camErrorType CamSetupCallbacks(GenericCameraHandle* hndl, sensorCallbacksListType* cbList)**

Set or change the existing sensor callbacks of a camera module (the interrupts callbacks are changed by the use of CamSetupInterrupts)

Parameters:

- hndl: pointer to the initialized camera handle
- cbList: list of sensor callback functions pointers (cbf)

Return parameter:

- CAM_SUCCESS

➢ **camErrorType CamSetupInterrupts (GenericCameraHandle* hndl, camIsrType managedInterrupt, u32 notifiedInterrupts, u32 clearedInterrupts, interruptsCallbacksListType* cbList, u32 interruptsLevel, u32 routeLineInterrupt, u32 routeFrameInterrupt)**

Set or change the existing interrupt events, interrupts levels and ids, callbacks for a camera module.

Parameters:

- hndl: pointer to the initialized camera handle
- managedInterrupt: the event (one only) on which CamGeneric will request a new buffer address by calling the user provided management callback and will update it in HW (values: see camIsrType)
- notifiedInterrupts: the event(s) on which CamGeneric will call the notification cbf (values: combined values of camIsrType type)

- clearedInterrupts: the event(s) which will be cleared in CamGeneric ISR handler (values: combined values of camIsrType type)
- cbList: the list of application callbacks to be called by CamGeneric on specific HW interrupts
- interruptsLevel: the level of priority to be assigned to the camera interrupt; values between 0-15 (15 not recommended)
- routeLineInterrupt: the id of the rerouted line interrupt, in case CamGeneric runs on Leon OS (values: 0 = use the LRT default id; IRQ_DYNAMIC_0 .. IRQ_DYNAMIC_11, seeDrvIcbDefines.h).
- For CIF receivers, this must have same value as the routeFrameInterrupt
- routeFrameInterrupt: the id of the rerouted frame interrupt, in case CamGeneric runs on Leon OS (values: 0 = use the LRT default id; IRQ_DYNAMIC_0 .. IRQ_DYNAMIC_11, seeDrvIcbDefines.h)

Return parameter:

- CAM_SUCCESS if the camera interrupts registers and callbacks were updated OK, other values if it failed

➢ **unsigned int CamGetFrameCounter (GenericCameraHandle *hndl)**

Get the counter of frames received inside the CIF receiver (for SIPP, this is always 0). The ISR management or notification must be enabled.

Parameters:

- hndl: pointer to the initialized camera handle

Return parameter:

- the received frames counter

The related data types for the offered API are given below:

camStatus_type (enum)	
Values	**Description**
CAM_STATUS_ACTIVE	Camera status active.
CAM_STATUS_HOT_STANDBY	Camera status hot standby.
CAM_STATUS_COLD_STANDBY	Camera status cold standby.

camErrorType (enum)	
Values	**Description**
CAM_SUCCESS	Operation performed successfully.
CAM_ERR_PARAMETERS_LIST_INCOMPLETE	The list of parameters sent to the current CamGeneric function contains NULL values for mandatory parameters.
CAM_ERR_CONFLICTING_ALLOC_FUNCTIONS	The list of parameters sent to the current CamGeneric function contains conflicting buffer allocation functions.
CAM_ERR_RECEIVER_ID_INCORRECT	The id of the sensor receiver inside myriad is not

camErrorType (enum)	
Values	**Description**
	correct.
CAM_ERR_MISSING_CALLBACK_OR_I2C_HANDLE	Neither an I2C handle nor an I2C configuration callback function were previously set for the current camera, so the sensor (re)configuration can not be performed.
CAM_ERR_I2C_COMMUNICATION_ERROR	An I2C communication error with the sensor happened.
CAM_ERR_MIPI_CONFIGURATION_ERROR	The parameters used for MIPI configuration were not corrects.
CAM_ERR_MIPI_PLL_INITIALISATION_ERROR	An error happened while initializing the MIPI: the PLL frequency could not be stabilized.
CAM_ERR_ROUTED_INTERRUPT_NOT_VALID	The configured dynamic IRQ, set in routeInterrupt parameter, is not correct.
CAM_ERR_STATUS_NOT_APPROPRIATE	The requested or current status of the camera does not permit the requested action.
CAM_ERR_FEATURE_TEMPORARY_NOT_IMPLEMENTED	This feature is currently not implemented but is foreseen to be in the next releases.
CAM_ERR_FEATURE_NOT_AVAILABLE_FOR_CURRENT_CONFIGURATION	This feature is not available for the requested HW configuration.

CamIsrType (enum)	
Values	**Description**
NO_CAM_ISR	No interrupt.
CIF_INT_EOL	CIF end of line interrupt.
CIF_INT_EOF	CIF end of frame interrupt.
CIF_INT_DMA0_DONE	CIF DMA0 done interrupt.
CIF_INT_DMA0_OVERFLOW	CIF DMA0 overflow.
CIF_INT_DMA0_UNDERFLOW	CIF DMA0 underflow.
CIF_INT_DMA0_FIFO_FULL	CIF DMA0 FIFO is full.
CIF_INT_DMA0_FIFO_EMPTY	CIF DMA0 FIFO is empty.
CIF_INT_DMA1_DONE	CIF DMA1 done interrupt (in case multi-planar transfer was requested).
CIF_INT_DMA1_IDLE	CIF DMA1 idle.
CIF_INT_DMA1_OVERFLOW	CIF DMA1 overflow.
CIF_INT_DMA1_UNDERFLOW	CIF DMA1 underflow.
CIF_INT_DMA1_FIFO_FULL	CIF DMA1 FIFO is full.

CamIsrType (enum)	
Values	**Description**
CIF_INT_DMA1_FIFO_EMPTY	CIF DMA1 FIFO is empty.
CIF_INT_DMA2_DONE	CIF DMA2 done interrupt (in case multi-planar transfer was requested).
CIF_INT_DMA2_IDLE	CIF DMA2 idle.
CIF_INT_DMA2_OVERFLOW	CIF DMA2 overflow.
CIF_INT_DMA2_UNDERFLOW	CIF DMA2 underflow.
CIF_INT_DMA2_FIFO_FULL	CIF DMA2 FIFO is full.
CIF_INT_DMA2_FIFO_EMPTY	CIF DMA2 FIFO is empty.
SIPP_INT_LINE_DMA_DONE	SIPP DMA done for line transfers.
SIPP_INT_FRAME_DMA_DONE	SIPP DMA done for frame transfers.
SIPP_INT_FRAME_LOST	SIPP exceptional event: one or more frames were lost.

callbacksListStruct (struct)		
Type	**Field name**	**Description**
sensorCallbacksListType	*sensorCbfList;	Pointer to a list of sensor dependent callback functions. Can be null if there are no custom sensor callbacks.
interruptsCallbacksListType	*isrCbfList;	Pointer to a list of callbacks to be called by CamGeneric ISR. Can be null if there is no need for ISR callbacks.

SensorCallbacksListType (struct)		
Type	**Field name**	**Description**
camErrorType	(***sensorPowerUp**) (I2CM_Device*, GenericCamSpec*, CamUserSpec*)	cbf to provide custom initialization of the sensor; MANDATORY if no I2C handler was provided at CamInit.
camErrorType	(***sensorStandby**) (I2CM_Device*, GenericCamSpec*, CamUserSpec*)	cbf to provide custom standby sequence of the sensor; MANDATORY if no I2C handler was provided at CamInit.

SensorCallbacksListType (struct)		
Type	**Field name**	**Description**
camErrorType	(***sensorWakeup**) (I2CM_Device*, GenericCamSpec*, CamUserSpec*)	cbf to provide custom wakeup sequence of the sensor; MANDATORY if no I2C handler was provided at CamInit
camErrorType	(***sensorPowerDown**) (I2CM_Device*, GenericCamSpec*, CamUserSpec*)	cbf to provide custom power down sequence of the sensor; MANDATORY if no I2C handler was provided at CamInit.

interruptsCallbacksListType (struct)		
Type	**Field name**	**Description**
frameBuffer*	(***getFrame**)(void)	cbf called by CamStart + CamGeneric ISR at the end of a frame data transfer, in order to provide a pointer to the next frame buffer to be filled by CamGeneric. In case the application does not need frame buffers management in CamGeneric, this cbf can be NULL. But in case this is set, then the line allocation cbf (getBlock) must be NULL.
frameBuffer*	(***getBlock**)(int)	cbf called by CamStart + CamGeneric ISR at the end of a line data transfer, in order to provide a pointer to the next line buffer to be filled by CamGeneric. In case the application does not need line buffers management in CamGeneric, this cbf can also be NULL. But in case this is set, then the frame allocation cbf (getFrame) must be NULL.
void	(***notification**)(int)	c bf called by CamGeneric ISR on any of the ISRs set in notificationInterrupts, to provide additional user functionalities. This is optional.

CamUserSpec (struct)		
Type	**Field name**	**Description**
eDrvMipiCtrlNo	**mipiControllerNb**:	The id of MIPI controller the sensor is connected to.
camRxId_type	**receiverId**:	The CIF/SIPP receiver block id.
u32	**sensorResetPin**:	The GPIO id used to reset the sensor, or value 0xFF if no reset is to be performed by the CamGeneric.
u32	**stereoPairIndex**:	The id of the I2C address inside the sensor header (0

CamUserSpec (struct)		
Type	**Field name**	**Description**
		for the first address, 1 for the second one).
u32	**windowRowStart**:	Vertical start position of the cropping window for the received sensor image.
u32	**windowColumnStart**:	Horizontal start position of the cropping window for the received sensor image.
u32	**windowWidth**:	Width of the cropping window for the received sensor image.
u32	**windowHeight**:	Height of the cropping window for the received sensor image.
camSyncSignalsType	***generateSync**:	Structure containing all the info needed to generate the synchronization signals for sensors connected on parallel interface (which are the GPIO's used for generating the HSYNC/VSYNC signals and the GPIO mode to be used for them).

drvMipiCtrlNo (enum)	
Values	**Description**
MIPI_CTRL_0	*MIPI Controller 0*
MIPI_CTRL_1	*MIPI Controller 1*
MIPI_CTRL_2	*MIPI Controller 2*
MIPI_CTRL_3	*MIPI Controller 3*
MIPI_CTRL_4	*MIPI Controller 4*
MIPI_CTRL_5	*MIPI Controller 5*

camRxId_type (enum)	
Values	**Description**
CIF_DEVICE0	CIF 0 receiver
CIF_DEVICE1	CIF 2 receiver
SIPP_DEVICE0	SIPP 0 receiver
SIPP_DEVICE1	SIPP 1 receiver
SIPP_DEVICE3	SIPP 3 receiver

camSyncSignalsType (enum)	
Values	**Description**
u8 vSyncGpio;	Vertical synchronization pin.
u8 vSyncGpioMode;	Vertical synchronization pin mode.
u8 hSyncGpio;	Horizontal synchronization pin.
u8 hSyncGpioMode;	Horizontal synchronization pin mode.

10.1.5 Configuration data

10.1.5.1 Static sensor configuration

The sensor must be described by a static header configuration in the CameraModules **folder**. The address to the configuration data is provided as parameter at CamInit() and is constant for the whole life of a camera (until stopped).

The data inside the header has to be structured according to the following types (as defined in **CameraDefines.h**):

GenericCamSpec (struct)		
Type	**Field name**	**Description**
u32	**frameWidth**:	Width of the frame generated by the sensor.
u32	**frameHeight**:	Height of the frame generated by the sensor.
u32	**hBackPorch**:	Number of PCLK clocks from the end of the horizontal sync pulse to the start of horizontal active period (only for sensors connected on parallel bus, 0 for the others).
u32	**hFrontPorch**:	Number of PCLK clocks from the end of the horizontal active period to the start of horizontal sync pulse (only for sensors connected on parallel bus, 0 for the others).
u32	**vBackPorch**:	Number of lines from the end of the vertical sync pulse to the start of vertical active (only for sensors connected on parallel bus, 0 for the others).
u32	**vFrontPorch**:	Number of lines from the end of active data to the start of vertical sync (only for sensors connected on parallel bus, 0 for the others).
mipiSpec	***mipiCfg**:	Pointer to a structure describing the MIPI characteristics (NULL for parallel bus sensors).
frameType	**internalPixelFormat**:	Format of the internal pixel storage.
u32	**bytesPerPixel**:	Number of bytes per pixel used for storage (equal to the number of received bpp, if there is no HW conversion in CIF/SIPP blocks).

GenericCamSpec (struct)		
Type	**Field name**	**Description**
u32	**idealRefFreq**:	Clock frequency to generate (for the parallel bus sensors only).
u32	**sensorI2CAddress1**:	I2C address of the unique sensor or of the left sensor (in case a pair of identical sensors is used).
u32	**sensorI2CAddress2**:	I2C address of the right sensor (in case a pair of identical sensors is used, 0 otherwise).
u32	**nbOfI2CConfigSteps**:	Number of I2C configuration steps to be performed; there must exist minimum 3 steps, the last 2 steps (N-1 and N) are reserved for the wakeup and standby steps, the first steps are the sensor power up steps.
I2CConfigDescriptor	***i2cConfigSteps**:	Pointer to an array of configuration steps.
u32	**regSize**:	Size in bytes of the sensor data register (the address is always on 2 bytes).
const u16	(***regValues**)[2]:	Pointer to an array of sensor registers values.

Related datatypes are:

mipiSpec (struct)		
Type	**Field name**	**Description**
eDrvMipiDataType	**pixelFormat**:	Format of the data on the MIPI interface (see eDrvMipiDataType).
u32	**dataRateMbps**:	Data rate (in megabyte per second) of one MIPI lane, with 50Mbps precision.
u32	**nbOflanes**:	Number of MIPI lanes used by the sensor.
eDrvMipiDataMode	**dataMode**:	Format of the data on the MIPI interface (see eDrvMipiDataMode).

I2CConfigDescriptor (struct)		
Type	**Field name**	**Description**
u32	**numberOfRegs**:	Number of sensor registers to be written by CamGeneric in the current configuration step.
u32	**delayMs**:	Delta time in milliseconds to wait after the current configuration step, before starting other configuration/ activation tasks.

frameType (enum)	
Values	**Description**
YUV420p	Planar 4:2:0 format.
YUV422i	Interleaved 8 bit.
YUV422p	Planar 8 bit.
YUV444i	–
RAW16	Any raw type > 8bits.
RAW8	Any raw type < 8bits.

eDrvMipiDataMode (enum)	
Values	**Description**
MIPI_D_MODE_0	MIPI Data Mode 0.
MIPI_D_MODE_1	MIPI Data Mode 1.
MIPI_D_MODE_2	MIPI Data Mode 2.
MIPI_D_MODE_3	MIPI Data Mode 3.

eDrvMipiDataType (enum)	
Values	**Description**
CSI_YUV_420_B8	CSI YUV 420 B8
CSI_YUV_420_B10	CSI YUV 420 B10
CSI_YUV_420_B8_L	CSI YUV 420 B8 L
CSI_YUV_420_B8_CSPS	CSI YUV 420 B8 CSPS
CSI_YUV_420_B10_CSPS	CSI YUV 420 B10 CSPS
CSI_YUV_422_B8	CSI YUV 422 B8
CSI_YUV_422_B10	CSI YUV 422 B10
CSI_RGB_444	CSI RGB 444
CSI_RGB_555	CSI RGB 555
CSI_RGB_565	CSI RGB 565
CSI_RGB_666	CSI RGB 666
CSI_RGB_888	CSI RGB 888

eDrvMipiDataType (enum)	
Values	**Description**
CSI_RAW_6	CSI RAW 6
CSI_RAW_7	CSI RAW 7
CSI_RAW_8	CSI RAW 8
CSI_RAW_10	CSI RAW 10
CSI_RAW_12	CSI RAW 12
CSI_RAW_14	CSI RAW 14

Although all the above data types are permitted for the MIPI transmission, the user also has to choose an internal pixel format (of frameType type). The conversion between the two data types is arbitrary. The only guaranteed correct conversion are downgrading RAW conversions (RAWx → RAWy, with x> y), for SIPP connected cameras only.

10.1.5.2 Dynamic sensor configurations

The user has to offer pointers to the sensor callback functions he want to be called on specific events.

Th is configuration is dynamic and can be offered by `CamInit()`, and `CamSetupCallbacks()` API functions.

If given in `CamInit()`, the sensor cbf must be included in a `callbacksListStruct` parameter, which is a wrapper over the sensor cbf and interrupt cbf.

if given by `CamSetupCallbacks()`, the sensor cbf can be offered directly through a `sensorCallbacksListType` parameter.

10.1.5.3 Dynamic user configuration

The user has to indicate some additional information which are not sensor dependent, but application dependent (HW/SW). This is dynamic configuration but is only provided through a **CamUserSpec** parameter in `CamInit()` API function, therefore is constant for the whole life of the camera (until stopped)

10.1.5.4 Static interrupts configuration

The static configuration for the interrupts is found in the **CamGenericPrivateDefines.h** header file of CamGeneric. The following defines can be updated to suite the project needs:

- CAMGENERIC_INTERRUPT_LEVEL // defaulted to 3
- ROUTED_IRQ_CIF0 // defaulted to IRQ_DYNAMIC_0
- ROUTED_IRQ_CIF1 // defaulted to IRQ_DYNAMIC_1
- ROUTED_IRQ_SIPP_LINE_ALL // defaulted to IRQ_DYNAMIC_2
- ROUTED_IRQ_SIPP_FRAME_ALL // defaulted to IRQ_DYNAMIC_4

MA2x5x configuration specificity

For the MA2x5x chipset, there is a new configuration to setup, for the private MIPI interrupt (see the note of chapter 10.1.2.3), in the file:

CamGeneric/arch/ma2150/leon/bm/include/CamGenericPrivateDefines_ma2150.h

- MIPI_INTERRUPT_LEVEL // defaulted to 2
- ROUTED_IRQ_MIPI // defaulted to IRQ_DYNAMIC_3

10.1.5.5 Dynamic interrupts configuration

By default, CamGeneric is handling the DMA done interrupts on end of frame, with local buffers management and no additional notification function, with clearing of all the existing interrupts when the ISR is handled. The interrupts are handled on the processor which is running the CamGeneric component, with interrupt level and rerouted interrupt id (if rerouting is needed) as statically configured in the `CamGenericPrivateDefines.h` file.

If this configuration does not match the application's needs, the application can dynamically change it, either by `CamInit()` and/or by `CamSetupInterrupts()` API functions:

- The **callback functions** to be called on interrupts (and implicitly the buffers management behavior and notification behavior): `getFrame/getBlock/notification cbf`.

  If given in `CamInit()`, the interrupts cbf must be included in the general `callbacksListStruct` wrapper, which also include the sensor cbf list**.**

  If given by `CamSetupInterrupts()`, the interrupts cbf list can be offered directly through the `interruptsCallbacksListType`.
- The **managed interrupt** / the **notified interrupts /** the **cleared interrupts** masks have to be provided in `CamSetupInterrupts()` API function.
- The **interrupts level** parameter is given by `CamSetupInterrupts()` API function.
- The **identifier** of the **routed line / frame** interrupts are also given by `CamSetupInterrupts()` API function.

10.2 Opipe

10.2.1 Introduction

The Opipe is an optimized Image Signal Processing (ISP) pipeline

It composed of a chain of SIPP hardware accelerator filters (there are no software filters in the pipeline). In the Opipe the output of each filter in the pipeline is connected directly to the next filter in the pipeline where it fills the local line buffer (if present) or is processed directly (without any copy to/from CMX memory).

This document further assumes reader is familiar with SIPP Hardware filters (see the relevant chapter in the Myriad 2 Databook, i.e., “Streaming Image Processing Pipeline Accelerators”).

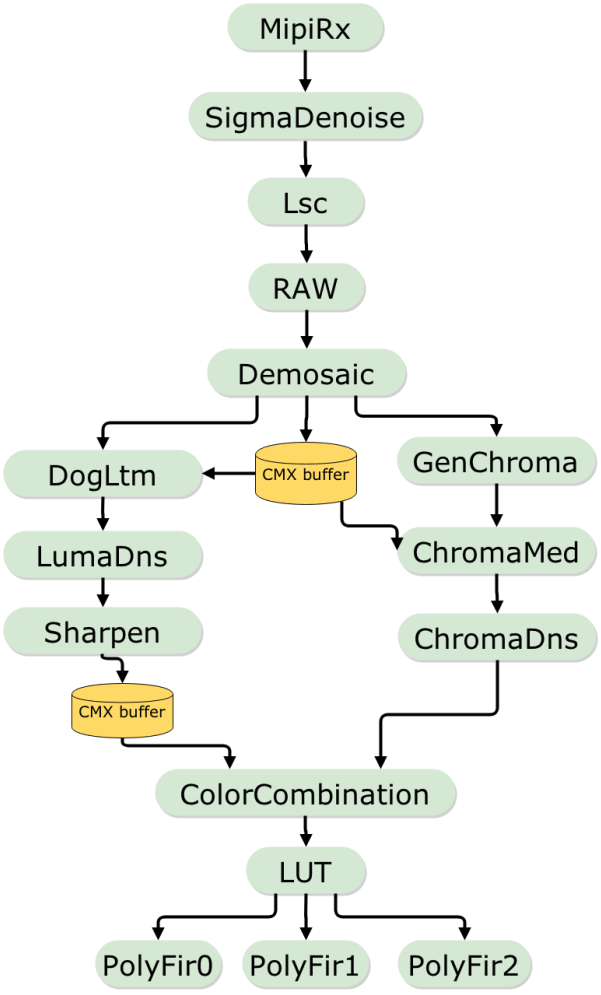


Figure 14: oPipe full graph

The input frames can be one of the following pixel formats:

- 8 bit
- 16 bit
- 32 bit
- Packed 10 bit

- Packed 12 bit

10.2.2 Description

The Opipe component is a Myriad 2 (Rev2-only) software module that allows users to build pipelines made exclusively of HW SIPP filters. An Opipe can be one of:

- Any standalone SIPP filter (even if not part of Opipe definition; e.g., Harris filter).
- Any contiguous Opipe subgraphs (including 1 up to all filters in Opipe graph).

The Opipe pipeline is interrupt driven (i.e. Leon responds to SIPP and CMXDMA interrupts). For performance reasons, Opipe uses SIPP IRQ-RATE feature that configures SIPP filters to fire interrupts every N lines (N is a power of 2) instead of every line. Typical IRQ_RATE value = 8 lines.

Multiple Opipe pipelines can run in parallel provided that they're not using common SIPP filter resources.

10.2.3 System Resources

- HW SIPP filters that participate to user pipeline.
- CmxDma using default mdk driver.

10.2.4 Terminology

Term	Description
Filter	Any SIPP HW filter.
Parent Filter	A **Filter** that is not driven by another **Filter** (instead is driven by DMA).
Leaf Filter	A **Filter** that has no other **Filter** consumers (instead is consumed by a DMA).
Source	Opipe internal Dma-entity that feeds a **Parent Filter.**
Sink	Opipe internal Dma-entity that consumes a **Leaf Filter.**
IBFL	Input Buffer Fill Level.
OBFL	Output Buffer Fill Level.

10.2.5 Typical Opipe Flow

Typically an Opipe is fed/consumed by CMXDMA or MipiRx blocks.

In a Opipe, top-parents and leaf-filters don't access the DDR full frame directly, but use smaller circular CMX buffers which are handled by CMXDMA (which is more efficient in accessing the DDR).

At setup (once per frame) Leon configures all filters (e.g., IBUF, OBUF, filter custom regs) then feeds the Parent-Filters several lines of data via CMXDMA. The number of lines is large enough to get the filter started (typical value: 16 lines).

When DMA transfer is complete, Leon updatesParent-Filters IBFLs accordingly to notify the Filters that input data is available. As a result (due to sufficiently increased IBFLs), filters start to produce data towards consumer-filters and gradually mark the input data as CONSUMED by decreasing IBFL-level.

In the IBFL-decrement IRQ, Leon triggers CMXDMA to feed new data in the recently released memory buffers, and so on until the entire input frame was fed.

Similarly for Leaf-Filters: as lines get produced, Dma Sinks copy data out to main buffer in DDR.

The remaining internal communication (within the Opipe graph) is handled by HW logic and internal line buffers (LLB = local line buffer).

10.2.6 DMA Drive

When driven by CMXDMA, Opipe uses 2 internal constructs to transfer the data: sources and sinks.

A Dma-**Source** copies data from the full buffer to smaller circular buffer and notifies (in the irq handler) the corresponding filter that new data has arrived (increments IBFL).

Similarly, a Dma-**Sinks** copy data from circular buffer to full DDR buffer and notify the corresponding filter accordingly (decrements OBFL).

Opipe component automatically attaches a source or sink to a CMX buffer, based on flags provided to **OpipeCfgBuff** routine. Some CMX buffers don't get a DMA associated as they get produced & consumed by HW blocks only (e.g. Sharpen-output, Debayer-Luma output).

10.2.7 Limitations

Opipe component limitations:

- The user **cannot** mix SIPP SW filters with SIPP HW filters.
- The user **cannot** use a SIPP HW filter multiple times, i.e. Opipe has exclusive access to the HW filter.

10.2.8 API

10.2.8.1 Core API

Function	Description
`OpipeReset`	Perform Opipe general setup (impacts all further defined individual Opipes); Initializes CMXDMA driver.
`OpipeInit`	Initializes pipe given as argument (clears & inits data structures).
`OpipeStart`	Starts pipe given as argument. This call is “NON-BLOCKING”.
`OpipeWait`	Waits till all Sinks of current pipe have completed copying the data back to DDR. This call is BLOCKING.
`OpipeDetCfg`	Computes OPIPE_CFG word based on filters enable mask and OPIPE hard-coded definition. This function is called internally from OpipeStart if not called externally by user. This routine is made public to allow setup factorization and reduce 1st start latency.

Function	Description
OpipeCfgBuff	Defines a new CMX buffer. All circular CMX buffers used by Opipe are allocated by user at application level for worst-case scenario.
OpipeSwLink	Defines a SW control link between a Parent and its Consumers. It assumes that only one of the consumers decrements Parent OBFL.
OpipeWaitForRawStats	If RAW stats are enabled, use this routine to make sure that RAW-stats writing to stats-buffer is completed. (RAW block starts to write gathered stats at EOF and lasts a certain amount of clock-cycles, depending on configured size and buffer location (e.g. DDR or CMX)).
OpipeSwLink	Defines a SW control link between a Parent and its Consumers. It assumes that only one of the consumers decrements Parent OBFL. This allows user to chain arbitrary filters together.

10.2.8.2 Utility Functions

Function	Description
OpipeTestIni	Performs general system initializations that are suitable for most tests (clock enables, sets Leon-L2 cache in write-through mode, lowers Leon Interrupt priority level).
defaultMipiTxLoopParams	Propose default mipiTx timing parameters.
cfgMipiTxLoopback	Configures certain sipp-mipi loopback mode. Possible paths: SIPP MIPI Tx[0] -> SIPP MIPI Rx[1] SIPP MIPI Tx[1] -> SIPP MIPI Rx[3]
startMipiTxLoopback	Start a mipiTx block (starts the timing generation).
OpipeDefaultCfg	Provides default configurations for filters involved in Opipe ISP construct.

10.2.8.3 Callbacks

User can enable following callbacks:

Callback Name	Description
Opipe::cbEndOfFrame	End of Frame callback: called when all SINKS in current Opipe finished copying data back to DDR.
Opipe::cbLineHit	Called when a source in pipe reached target line (note: line number needs to be quantized to be a multiple of IRQ_RATE). One can set the desired callback point via “Opipe::targetLine” array.

Callback Name	Description
`Opipe::cbPreStart`	Pre-start callback to allow user to patch obscure Opipe ISP assumptions.

10.2.8.4 Application Helpers

Typical ISP Opipe helpers are added to OpipeApps.c/h. These files are not a core part of Opipe component, but are very likely to be used in principal Opipe applications.

Helper Name	Description
`OpipeCreateRx`	Initializes current Opipe for MipiRx->DDR transfer.
`OpipeCreatePP`	Initializes current Opipe for Pre-Processor operation. This pipe has mipiRx, Sigma, Lsc and RAW blocks in the pipeline.
`OpipeCreateFull`	Creates a Full-Isp Opipe configuration (filters between SIGMA and POLYFIR0/1 inclusive, as per Opipe diagram).
`OpipeCreateMain`	Creates a Main-Isp Opipe config (filters between DEBAYER and POLYFIR0/1 inclusive, as per Opipe diagram).
`OpipeCreateLumaMono`	Creates a mono ISP pipeline with Luma input and Luma output.
`OpipeCreateBayerMono`	Creates a mono ISP pipeline with Bayer input and Luma output.
`OpipeSetSizeMF`	Used to change input resolution of a MF (Main or Full) pipeline. **NOTE:** Output resolution is dictated by PolyFir N, D params.
`OpipeSetDoglCfg`	Allows pipe reconfiguration around DOGL to trade quality for performance. Available modes: DOGL_ON_F16_CMX : both read clients are used, luma format=fp16 DOGL_ON_U8F_STREAM : a single read client is used, luma format = u8f DOGL_OFF : DOGL block bypassed completely.

10.2.8.5 Standalone Sigma case study

Main steps for configuring an Opipe made of a single filter (case study here: SIGMA denoise) are:

1. System initialization. This is project/application specific, e.g.

```
initClocksAndMemory();
```

2. Opipe component initialization:

```
OpipeReset();
```

3. Application specific Opipe initialization and configuration:

```
OpipeInit(p);
appSetParams02();
p->enMask = 1<<SIPP_SIGMA_ID;
p->width  = IMG_W;
p->height = IMG_H;
p->cfg[SIPP_SIGMA_ID] = …
```

4. CMX buffer configuration. Since this is a single-filter pipeline, both Input and Output filters are attached to Sigma filter:

```
pIn = OpipeCfgBuff (p, SIPP_SIGMA_ID, D_DMA|D_IN,
                    (uint32_t)iCircBuffCmx2, …
pOut = OpipeCfgBuff (p, SIPP_SIGMA_ID, D_DMA|D_OUT,
                     (uint32_t)oCircBuffCmx2, …
```

5. Full image buffer assignment:

```
pIn->ddr.base = (uint32_t)iBuf;
pOut->ddr.base = (uint32_t)oBuf;
```

6. Start processing frame and wait for completion:

```
OpipeStart(&p2);
OpipeWait(&p2);
```

10.3 JPEG Encoder

10.3.1 Overview

This report gives a brief outline of how JPEG encoder works and how to use it.

JPEG encoder component compresses .yuv image files into .jpg files. It supports YUV420, YUV422, and YUV444 planar as input.

10.3.2 Architecture and implementation

JPEG encoder functionality is split between LeonOS and a configurable number of Shaves (from 1 to 12 shaves).

Leon is doing the scheduling part and also assembles in the right order output data which is produced by shaves.

Each shave has three local buffers for each plane of input data. Input data is copied from DDR into local buffers using DMA transfer.

10.3.2.1 Implementation

In the Leon's CMX slice is declared an array of 24 (the maximum number of shaves * 2) output buffers. This is because double or multiple buffering is needed for output in order to overlap shave processing and data copying back to DDR. Shaves are working asynchronous in order to increase efficiency.

Input image is split into macroblocks and each shave processes the same number of macroblocks. Output of HuffmanEncoder is given in bits not bytes. Aligning the output from consecutive processed chunks is expensive. So in order to avoid the alignment of bits each output chunk will be found between two reset markers.

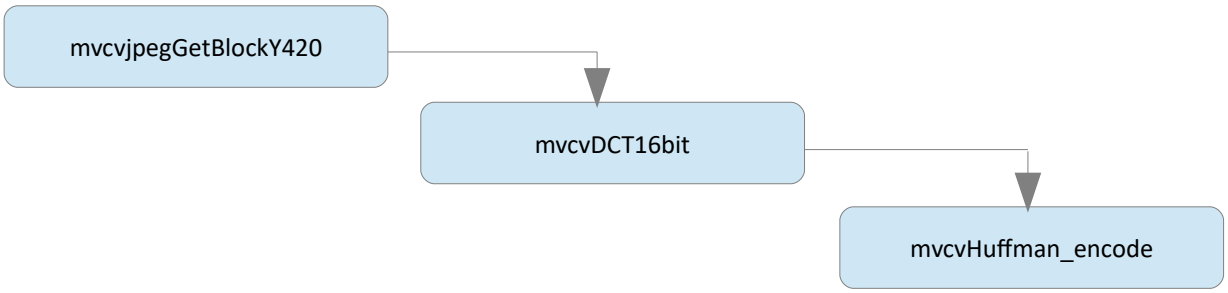
10.3.2.2 Memory allocation

CMX : Local input buffers for shaves (60K for each).
Shave output buffers shared with Leon (total size: 7K * SHAVE_MAX_NO * 2).
DMA descriptors.

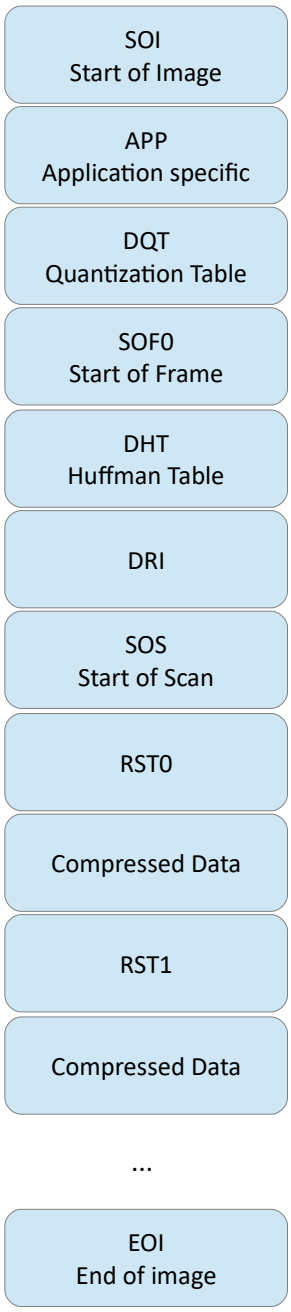
DDR : Input and output buffers are expected to be in DDR.

10.3.2.3 Assembly optimized functions

Major processing steps were written in assembly.



10.3.2.4 Output JPEG structure



10.3.2.5 Restrictions

Because the macroblocks size for P420 width should be N*16 and for P422 and P444 N*8. In order to achieve good performance if width is higher then a certain value (1888 if image is P420 for example) width/ blockW (where blockW is 8 for P444 and 16 for P420 and P422) should not be a prime number.

10.3.3 Usage and Results

10.3.3.1 JPEG encoder API

The JPEG encoder function prototype is the following:

```
u32 JPEG_encoder(frameBuffer imgInfo, u8 *output, u32 shvNo, int
jpegFormat)
```

where:

- `imgInfo` is a specific MDK frameBuffer structure – we are concerned in this case only about pointers to each plane of the YUV input image, width and height of the image.
- `output` is a pointer to the output buffer.
- `shvNo` represents the number of shaves on which JPEG encoder will run (starting from shave 0).
- `jpegFormat` can take one of the following values: JPEG_420_PLANAR, JPEG_422_PLANAR or JPEG_444_PLANAR.

10.3.3.2 Performance

Image format	Shave number	Cycles / pixel
P420	1	1.94
	12	22.56
P444	1	5.22
	12	57.36

NOTE: These values have been computed for a 12Mp image.

10.3.3.3 Quality

PSNR is: 29.027 for Y, 46.501 for U, 40.552 for V.

NOTE: These values have been computed for a 2104x1560 image.

10.4 Power Manager

The power manager component allows the user to configure different power modes and transparently switch between them.

10.4.1 Description

The power manager is implemented as a RTEMS kernel driver. The interface allows any task or driver to register itself with the power manager by opening the device, register a set of callbacks that would notify the task about power changes and read and modify the system's global power state.

Any task or driver that has not registered with the power manager will not be able to influence the system decisions related to power states or to have personalized actions while these actions happen.

The callbacks allow the user to control the power states in the following moments:

- Request: requests from the task to change power mode; each task is allowed to deny this request.
- Granted: in the case that all registered tasks approved the change of the power state that the system changes power mode.
- Cancel: in the case that even one of the registered tasks denies the request, the registered tasks that had previously approved the power change get notified that the request has been canceled.
- Restore: after the system returns from the changed power state, all tasks are informed that the default power state will get restored.

Currently the power manager supports only Active and Low Power modes as valid power states. Please refer to Myriad 2 latest databook for further details on the different Power modes available.

10.4.2 Active mode

If the power manager is linked with an application, it replaces the idle task that RTEMS provides with a bigger function that takes into consideration the current power mode.

If no explicit power mode is set by the user, the default behavior of the system is to execute a function that corresponds to the ACTIVE state, which is identical with the RTEMS' idle loop implementation.

Active mode is the default state all processes start with and the state all processes return to. This means that if Low power mode is executed, after the system exits Low power mode, its state will reset to Active, and it would be the user's responsibility to set it again, if this is needed.

10.4.3 Low power mode

When the Myriad 2 SoC is entering in low power mode, all power domains are being turned off, except for CSS and RETENTION power domains. The Leon processor is executing from the LOS L2 cache in locked mode and the LOS processor enters in sleep state.

Currently the following settings are configurable regarding a user-defined power state:

- Wake-up source.
- Wake-up configuration.
- Clock source.
- Clock configuration.

For now, only GPIO configuration is accepted as a valid wakeup source. In this case, the wakeup configuration will be consisted by a specified GPIO number, that will trigger the system wakeup when set to its positive level.

Regarding the clock configuration, to achieve minimum power usage while in low power mode, the 32KHz external clock has to be used, however this would introduce bigger latency while the system restore its power state.

For this reason, we allow the user to choose between the 32 KHz clock and the OSC1 clock (usually 12 MHz) as sources for the low power mode. If the OSC1 clock is chosen, the user can configuring a numerator and a denominator for this clock, to achieve a fraction of its frequency.

The callbacks functions are called before the system enters in low power mode and after it returns by the system's idle task, thus the user defined implementation should be non-blocking and if possible low overhead, similarly as if it is called from an interrupt context.

10.4.4 Interface

The Power Manager offers a POSIX API, thus after the driver registration and initialization, it should be accessed only by `open()`, `close()`, and `ioctl()` instead of custom `rtems_io_` functions.

10.4.4.1 Register driver

A task should use the registration function **rtems_io_register_driver()** with the **osDrvPwrManagerTbl** as parameter to register Power Manager with RTEMS device table. On success RTEMS_SUCCESSFUL is returned.

10.4.4.2 Initialize driver

A task should use the initialization function **rtems_io_initialize_driver()** with a predefined array of (non-active) power modes; The array should have at least one valid entry and at most POWER_MANAGER_MAX_IDX entries (16). For the driver to determine its size, It should be terminated with an entry that contains the “invalid” mode POWER_MANAGER_MODE_NONE.

On success, RTEMS_SUCCESSFUL is returned.

On failure:

- RTEMS_INCORRECT_STATE is returned if the input power mode array is invalid (containing unknown modes or the mode MODE_ACTIVE).
- RTEMS_TOO_MANY is returned if the given array has more than POWER_MANAGER_MAX_IDX entries.
- RTEMS_NOT_CONFIGURED is returned if the given array is empty.

10.4.4.3 Open

Every tasks needing power management personalization should call **open().**

The Power Manager driver is allowed to be opened at most POWER_MANAGER_MAX_USERS (16) times. A successful open returns a valid POSIX file descriptor and gives access to the task to the internal functions of the power manager. If the device is already opened MAX_USERS times, open fails with -1 and errno is set.

10.4.4.4 Close

On **close()**, all callbacks registered from the caller task are removed.

10.4.4.5 IO control interface

The IO control interface provides a set of four operations that allow the user to add and delete its own callbacks and read and switch the system's power state.

For the callback related IO control operations, the following two functions are implemented:

- PWR_MANAGER_ADD_NOTIFICATION_CALLBACKS
- PWR_MANAGER_DEL_NOTIFICATION_CALLBACKS

Both functions receive as argument a data structure that contains the list of callbacks and a void pointer that will given as arguments to the callbacks, if used. For the ADD operation, the user needs to provide a valid callback table, and a void* argument the registered callbacks will be called with. For the DEL operation, the user's input is ignored. For both functions, RTEMS_SUCCESSFUL is returned on success. If the driver has not registered callbacks and tries to delete them, RTEMS_NOT_CONFIGURED will be returned.

It has to be noted that currently the remaining system drivers are not integrated into the power manager and the user is responsible modifying any running tasks that need to use the power manager by himself.

For the power state related IO control operations, the following two functions are implemented:

- PWR_MANAGER_SWITCH_POWER_STATE

- PWR_MANAGER_GET_CURRENT_POWER_STATE

Both functions receive as argument a data structure that contains the a mode field that the user wants to switch to, or get. For the switch operation, the index of a valid power state is required, while for the get operation the current power state is stored in the mode field. In the case there is no configured mode, -1 will be returned as state, while -1 can be used from the user to cancel a previously requested power state.

10.5 Pipe Print

The Pipe Print component allows the user to use debug pipe instead of UART for printf debug text messages to be displayed in the debugger from Leon OS and Leon RT processors.

10.5.1 Description

The pipe print component implements a function that provides a circular memory buffer to store printf messages, this buffer is then read by the debugger and displayed in the console output of movidebug2, allowing debug messages to be displayed via JTAG connection only. In order for this to work the debugger must be configured to read this buffer. This is done automatically by MDK build system, by providing the appropriate debug script.

Another major advantage of this component is that it provides debug messaging capabilities that do not require HW interrupts that take a lot of time and thus can halt RTEMS tasks for as long as 0.5 seconds effectively allowing real-time code execution while having debug messages active.

10.5.2 How it works

If the pipe print component is linked with an application, it replaces the function that provides message to UART transmission with a faster message to memory storing solution. By providing a definition at compile time of the PIPEPRINT_SECTION define the user can configure in which memory section the circular memory buffers for TX and RX will be created, the default value of this define is `#define PIPEPRINT_SECTION ".ddr_direct.data"`. It is important to use un-chaced memory section.

The size of the memory buffer can be configured the same way as the section by defining PIPEPRINT_SIZE, the default value for this memory buffer is `#define PIPEPRINT_SIZE (50*1024)`. 2 memory buffers will be created. One for TX messages called `mvConsoleTxQueue` that serves as the buffer when the application will store the messages that need to be displayed in the debugger console and the other for RX messages: `mvConsoleRxQueue` where the application can read information coming from the debugger.

10.5.3 Interface

To use the PIPE PRINT component the user build it for the processor where he wishes to use debug printf-s. This is best accomplished using the MDK build system variables:

```
ComponentList_LOS    +=  PipePrint
ComponentList_LRT    +=  PipePrint
```

No other code changing is necessary to use the component.

Using Makefile variables the user can customization the memory section and the buffer size.

First define the following variables in the application Makefile:

```
PIPEPRINT_SIZE = (10*1024)
```

```
PIPEPRINT_SECTION = ".cmx_direct.data"
```

Then add them as a define to the compiler options for LOS:

```
CCOPT      += -D'PIPEPRINT_SIZE=$(PIPEPRINT_SIZE)' -
D'PIPEPRINT_SECTION=$(PIPEPRINT_SECTION)'
```

and for LRT if needed:

```
CCOPT_LRT += -D'PIPEPRINT_SIZE=$(PIPEPRINT_SIZE)' -
D'PIPEPRINT_SECTION=$(PIPEPRINT_SECTION)'
```

Each instance of this component allocates a separate circular buffer for TX and RX channels having the same size. There are 2 possible instances of this component one for Leon OS and another for Leon RT, each allocating separate TX and RX buffers.

10.5.4 Limitations

The PipePrint component can not be used for SHAVEs. Therefore, in case debug messages need to be sent from the SHAVEs, it is still necessary to have an `uart` connection.

11 Application Profiling

11.1 Profiling information

The SDK tools include a function profiler, a sampling profiler, as well as the ability to add trace profiling to one's application. Information on using all of these can be found in the moviProf document delivered in the MDK bundle as well as in the `common/components/profiler` component Readme and ASCIIDOC documents.

12 References

[ref1] **SPARC Architecture Manual, version 8**
http://gaisler.com/doc/sparcv8.pdf

[ref2] **Leon IP Core Manual**
http://www.gaisler.com/products/grlib/grip.pdf

[ref3] **Myriad Platform Design Databook**
(released by Intel Movidius under NDA)

[ref4] **SHAVE Instruction Set Manual**
(released by Intel Movidius under NDA)

[ref5] **Movidius Debugger User Manual**
moviDebug.pdf is released as part of the Myriad Development Kit (MDK).

[ref6] **MDK Getting Started Guide**
MDK-GettingStarted.pdf is released as part of the Myriad Development Kit (MDK).
This document describes how to build the first application, and also details source locations of key components.

[ref7] **MA2x5x SIPP User Guide**
Released through www.movidius.org

[ref8] **RTEMS Wiki**
http://www.rtems.org/wiki/index.php/Main_Page

[ref9] **RTEMS applications C user's guide**
https://docs.rtems.org/branches/master/user/index.html

[ref10] **RTEMS POSIX API user's guide**
https://docs.rtems.org/releases/rtems-docs-4.11.2/posix-users/index.html

[ref11] **RTEMS POSIX 1003.1 compliance guide**
https://docs.rtems.org/branches/master/posix-users/index.html

[ref12] **RTEMS BSP and device driver development guide**
https://docs.rtems.org/releases/rtems-docs-4.11.2/bsp-howto/index.html

[ref13] **MDK Release Notes (version xx.xx.x)**
MDK_Release_Notes_xx.xx.x.pdf is released as part of the Myriad Development Kit (MDK).