Horizon 2020 PROGRAMME *ICT-01-2014: Smart Cyber-Physical Systems*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 643924

D2.3

Application processor description

1. DOCUMENT INFORMATION

Deliverable Number	D2.3
Deliverable Name	Application processor description
Authors	Alireza Dehghani (Movidius), Aubrey Dunne (Movidius), David Moloney (Movidius), Oscar Deniz (UCLM)
Responsible Author	Alireza Dehghani (Movidius) e-mail: alireza.dehghani@movidius.com phone: +353 86 1502011
WP	WP2
Nature	O
Dissemination Level	PU
Final Version Date	3.03.2016
Reviewed by	O. Deniz (UCLM)

2. DOCUMENT HISTORY

Person	**Date**	**Comment**	**Version**
Alireza Dehghani	29.02.2016	Initial version	0.1
O. Deniz	3.03.2016	Review	0.2

3. TABLE OF CONTENTS

4. ABSTRACT

Myriad 2 is a multicore, always-on system on chip that supports computational imaging and visual awareness for mobile, wearable, and embedded applications. The vision processing unit incorporates parallelism, instruction set architecture, and micro-architectural features to provide highly sustainable performance efficiency across a range of computational imaging and computer vision applications, including those with low latency requirements on the order of milliseconds. In this document, Myriad2 as the heart of the EoT platform will be described in detail.

5. MYRIAD2

[redacted] Introduction

Computer vision is beginning to transition from the laboratory (PC-based) to the real world in a host of applications that require a new approach to supporting the associated power, weight, and space constraints (embedded-based). Although PC-based strategy is convenient for prototyping new algorithms, it has no direct path to real world applications such as mobile phones, tablets, wearable devices, drones, and robots, where cost, power, and thermal dissipation are key concerns. Despite the work done on transitioning such applications using conventional mobile phone processors, such processors still present issues to the application developer in terms of real-time and computational requirements. Marshalling these diverse resources is possible, but it puts a major strain on both the programmer and the application processor fabric. As an example, dissipating 10 W in a drone might not be a thermal issue, but it leaves less power available for flying, whereas in a mobile phone 3 to 4 W can be dissipated on average before the phone becomes literally too hot to handle.

Clearly, across all of the possible applications a better solution is required to evolve to truly capable computer vision systems with intelligent local decision making for autonomous robots and drones as well as truly immersive virtual reality experiences. The vision processing unit (VPU) solution presented by Movidius uses a combination of low-power very long instruction word (VLIW) processors with vector and SIMD operations capable of very high parallelism in a clock cycle, allied with hardware acceleration for key image processing and computer vision kernels, backed by a very high bandwidth on-chip multicore memory subsystem. Particularly, the Myriad2 VPU aims to provide an order of magnitude higher performance efficiency than the Myriad 1, allowing high-performance computer vision systems with very low latency to be built while dissipating less than 1 W.

This deliverable describes the functionality and use of the Myriad 2 multiprocessor SoC family. This document provides a general overview of the Myriad 2 devices and also introduces some other relevant documentation.

[redacted] Myriad 2 family overview

The Myriad2 system on chip (SoC) device family offers twelve SHAVE vector processors with two 32-bit RISC (LEON) to provide exceptional performance efficiency and flexibility. A brief overview of the Myriad2 common features is 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 Myriad2 family consists of the following two series:

- MA2100
- MA2x5x – MA2150 / MA2155 / MA2450 / MA2455

5.2.1. Myriad2 - MA2100

MA2100 is the first member of the family with the following specific features:

- 500 MHz system clock
- 128 MB of in-package stacked LP-DDR2 @ 500 MHz
- USB 2.0 operation
- 90 MPixels/SIPP ISP pipeline throughput
- 5 simultaneous camera support

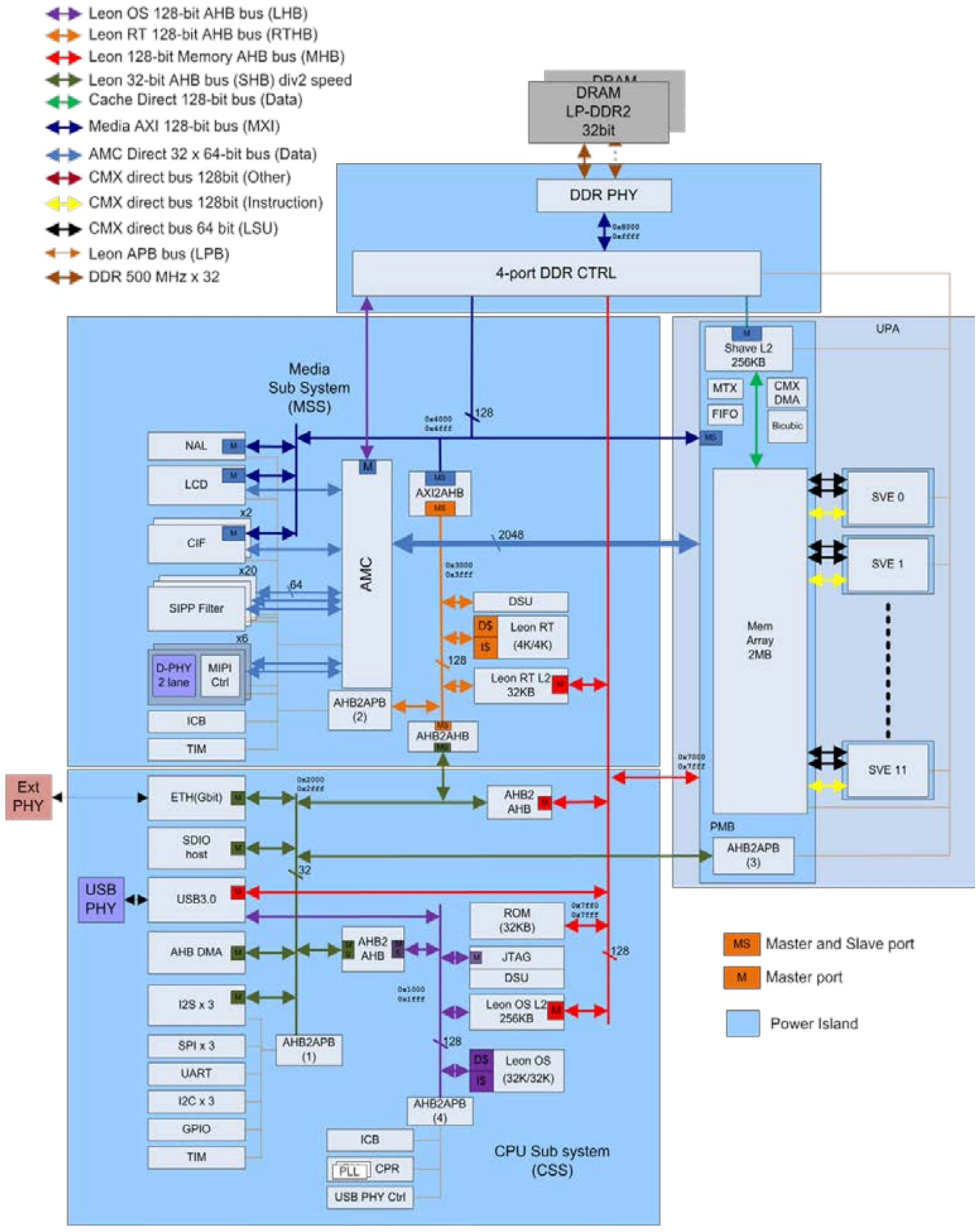

Figure 1: MA2100 Architecture Diagram – Block level

The MA2100 architecture block diagram can be found on Fig. 1.

5.2.2. Myriad2 - MA2x5x series

MA2x5x is the second series of the Myriad 2 family with four members (MA2150/MA2155/MA2450/MA2455) and the following specific features:

- 600 MHz system clock
- USB 3.0 operation
- Dual voltage SDIO 1.8 V & 3.3 V
- USB boot mode
- IQ and performance improvements in SIPP ISP pipeline – 600 MPixels/s throughput
- On-die temperature sensors
- 6 simultaneous camera support
- 128 MB of in-package stacked LP-DDR2 @ 533 MHz (MA215x)
- 512 MB of in-package stacked LP-DDR3 @ 933 MHz (MA245x)
- Secure boot mode (MA2x55)
- Low power state improvements

The MA2150 architecture block diagram can be found on Fig. 2.

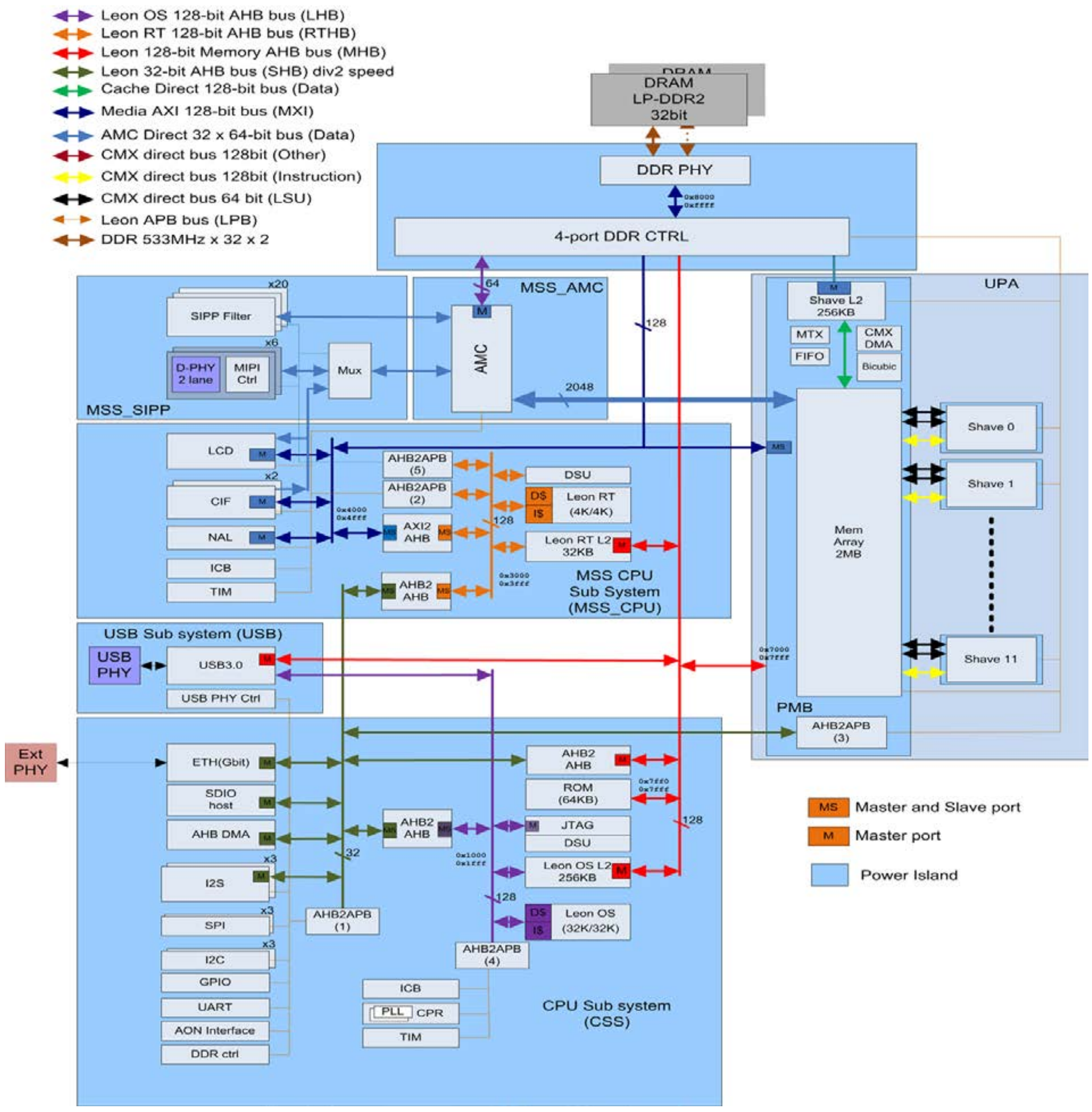

Figure 2: MA2150 Architecture Diagram – Block level

[redacted] Myriad2 Block level architecture overview

The block diagrams presented in the previous section 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).

5.3.1. The Media Sub System (MSS)

The MSS is the architectural unit designed for allowing external connections with imaging devices (camera sensors, LCDs, HDMI controllers etc.) as well as allowing use of the HW filters available in Myriad 2. As such it is comprised by the MIPI, LCD, CIF interfaces, the SIPP HW filters and well as the AMC block which enables connections between these and CMX (SRAM) memory.

Coordinating frame input and controlling the pipelines set in place usually require some coordination effort. As such, the Myriad 2 platform offers the Leon RT RISC as part of the MSS. Leon RT (LRT) is a RISC processor with a fair amount of L2 cache memory (32 KB). Leon RT is only one arbiter away from any 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.

5.3.2. 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 control unit of this block is the Leon OS (LOS) RISC processor, but in this block the Leon owns much bigger L1 (32 KB) and L2 (256 KB) caches, which allows to put a modern RTOS on it. This block also offers an AHB DMA engine for more optimal data transfer via the external peripherals. Besides handling the external interfaces and communication Leon OS could also control SHAVE processors imaging algorithms.

5.3.3. The Microprocessor Array (UPA)

The UPA is the design unit in Myriad 2 holding the 12 VLIW SHAVE vector processors, the 2 MB CMX SRAM memory and a few other blocks from which we list: the specialized DMA engine, the 256 KB L2 cache memory available to the SHAVE cores.

This design unit’s main purpose is to provide support for customized code required by many imaging or computer vision applications as well as any other general computation intensive algorithms.

The CMX memory itself will be described in greater detail in a further section but the author would like to highlight at this stage one aspect which is better understood in conjunction with the block diagram: each SHAVE processor has preferential ports into a 128 KB slice of the CMX memory. As such, 12x128 KB = 1536 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.

[redacted] Myriad 2 programming paradigms

5.4.1. Standard programming paradigm

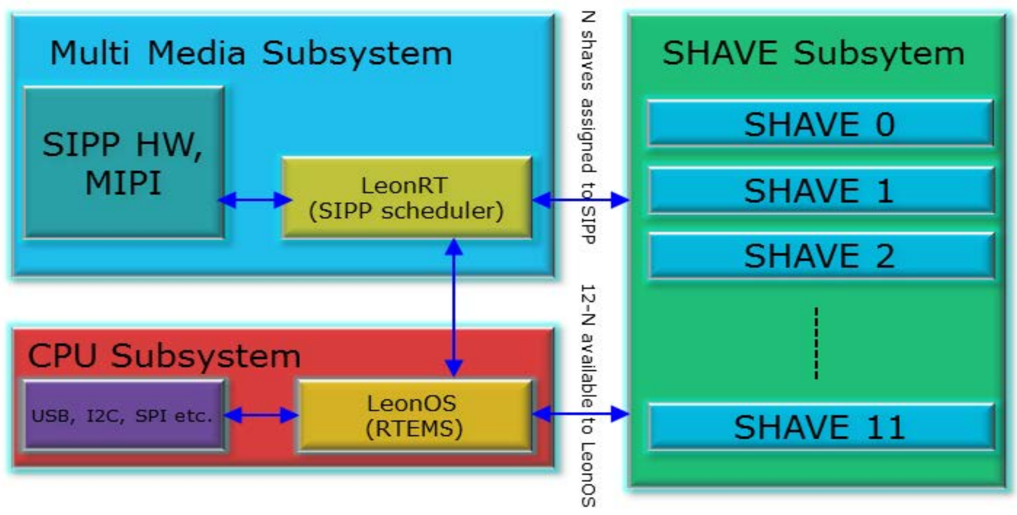

Figure 3: Standard programming paradigm

The standard programming paradigm for Myriad 2 involves using RTEMS running on LeonOS and the SIPP scheduler on Leon RT. 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 HW filters and exterior interfaces with a low footprint so as to ensure LeonRT optimized utilization.

The SIPP used number of SHAVEs is configurable, so any extra number of SHAVEs not used for line based pipelines will remain free to be used by the RTEMS operating system running on Leon OS for various other purposes including (but not limited to)

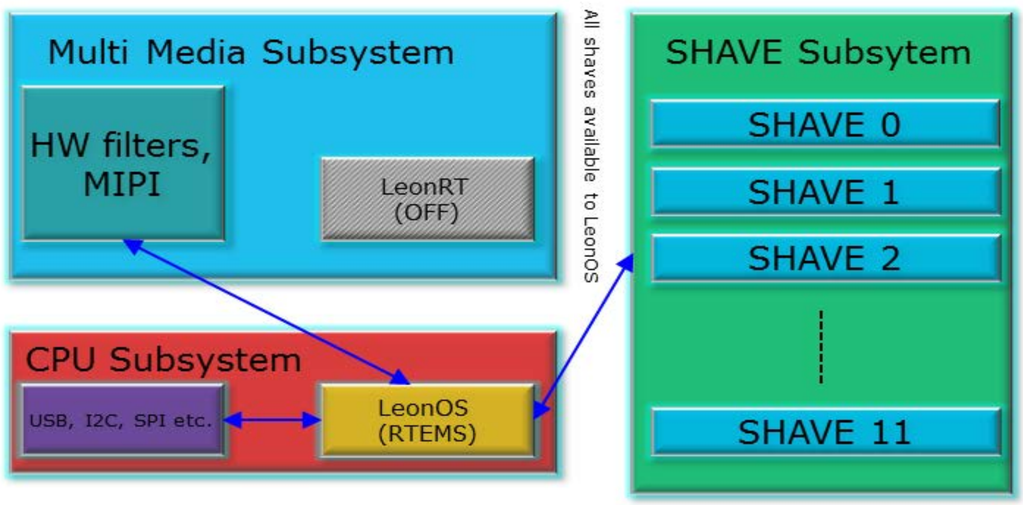

Figure 4: One Leon programming paradigm

computer vision algorithms.

5.4.2. The 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.

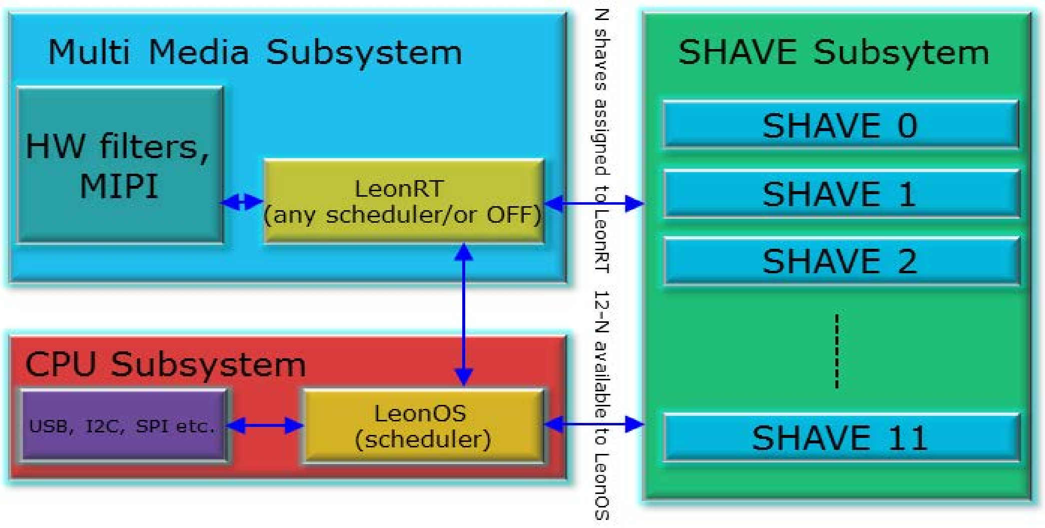

Figure 5: Bare metal programming paradigm

5.4.3. 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.

[redacted] Other relevant documentation

The following supporting documentation is available in addition to this guide. These documents are referenced elsewhere 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.

[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 Movidius under NDA)

[ref4] SHAVE Instruction Set Manual

(released by Movidius under NDA)

[ref5] Movidius Debugger User Manual

moviDebug.pdf is released as part of the Myriad Development Kit (MDK)

[ref6] Movidius 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] MA2100 SIPP User Guide

MA2x5x SIPP User Guide

Released through www.movidius.org

6. LEON

[redacted] 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].

[redacted] Overview

Two LEON4 cores are used. They will be referred to in this document as LeonOS and LeonRT. Their purposes/usage was described in the introductory section. 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

[redacted] Programming details

6.3.1. Addressing Scope

Both LeonOS and LeonRT processors have complete access to the flat Myriad 32-bit address space. However, 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.

6.3.2. 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].

6.3.3. Cache Coherency

Separate instruction and data caches are available in the LEONs 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 the 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 Fig. 7 for a diagram about the cache organization in Myriad 2.

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

6.3.4. 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.

6.3.5. Endianness

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

6.3.6. 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 registers 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.

6.3.7. 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.

6.3.8. 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

[redacted] 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 SHAVEs.

Despite LeonOS being capable of numeric computation, its capabilities are significantly inferior to those of the SHAVEs. 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 SHAVEs.

6.4.1. 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.

6.4.2. 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.

[redacted] 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.

6.5.1. heap

The default heap is set to 6 KB 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.

6.5.2. printf

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

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

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 `printfs` from their application.

7. SHAVE

[redacted] 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.

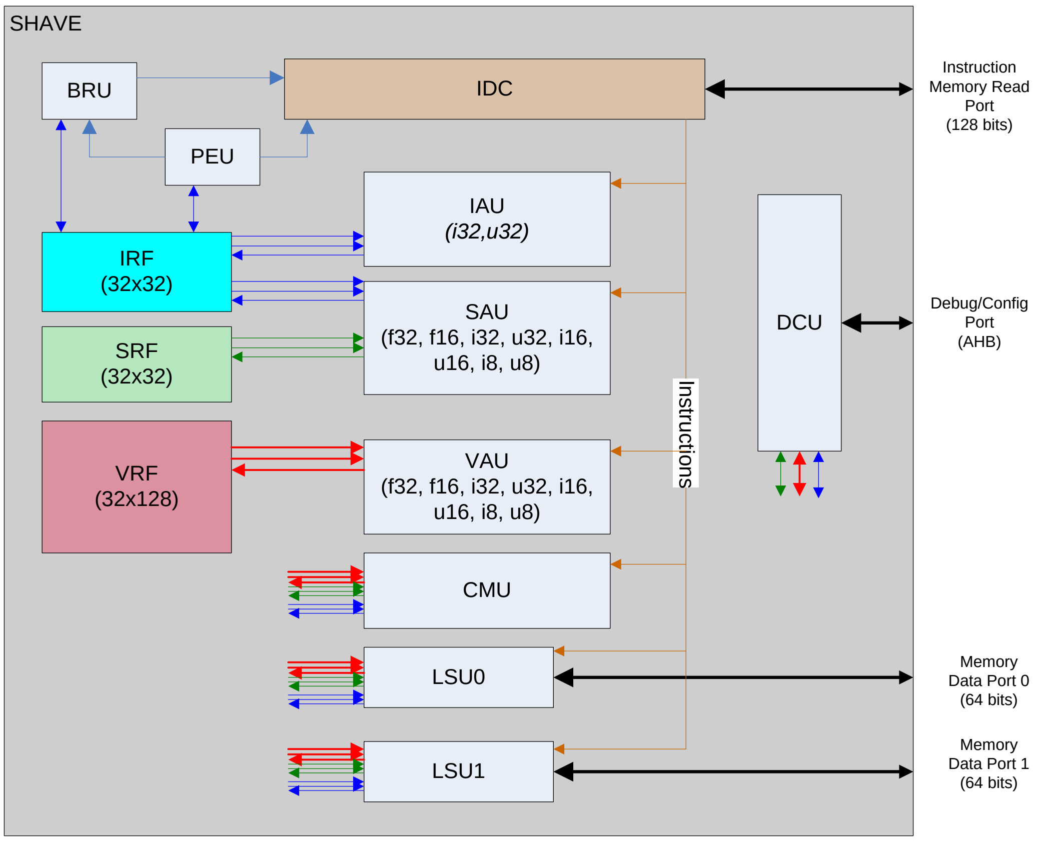
SHAVE
BRU
IDC
PEU
IAU (i32,u32)
IRF (32x32)
SAU (f32, f16, i32, u32, i16, u16, i8, u8)
SRF (32x32)
VRF (32x128)
VAU (f32, f16, i32, u32, i16, u16, i8, u8)
CMU
LSU0
LSU1
DCU
Instructions
Instruction Memory Read Port (128 bits)
Debug/Config Port (AHB)
Memory Data Port 0 (64 bits)
Memory Data Port 1 (64 bits)

Figure 6: 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.

7.1.1. 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.

7.1.2. 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.

7.1.3. IAU (Integer Arithmetic Unit)

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

7.1.4. 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.

7.1.5. 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.

7.1.6. 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.

7.1.7. 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.

7.1.8. 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.

7.1.9. PEU (Predicate Execution Unit)

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

[redacted] Running SHAVE code

7.2.1. 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.

7.2.2. 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 bits.

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.

[redacted] Standard Library support

7.3.1. 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

7.3.2. 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

7.3.3. 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

[redacted] Mutexes

7.4.1. 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
- SHAVEs 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`).

7.4.2. Recommended usage

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

[redacted] SHAVE Assembler

7.5.1. 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.

7.5.2. 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.

7.5.3. 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.

[redacted]Typical SHAVE usage

The SHAVE processor is good at performing intensive computational tasks. Typically, data would be buffered from DDR to CMX memory chunk by chunk and it would be processed there, and then moved back to output DDR if needed. The `VideoEffectsHDMI` example delivered along with the MDK shows such processes happening on video frames.

In order to achieve the best results while minimizing development time, the best usage of the Myriad processors would be 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 sections, it is preferable to let the LEON processors handle the treating of various interrupts while the SHAVE is performing its processing of data.

7.6.1. 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 ISP functionality. SIPP is described in the MA2100 and MA2x5x SIPP User Guides. Please see section 15.

8. MEMORY OVERVIEW

[redacted] Introduction

8.1.1. Overview

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

8.1.2. Memory areas

Memory	Size	LEON Access Cost	SHAVE access cost
CMX	2 MB	Low	Low
DDR	128 MB (MA2100)	High Low when data cache hit	Low for L1 cache hit Moderate when L2 hit High for random access

[redacted] Memory Map

8.2.1. Memory Map table (abridged)

Peripheral	Addresses	Size
Camera 1 (CIF)	30FA0000	–
DDR	80000000	128MB (MA2100)
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`	–

1.1.1 Memory Map Notes

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

- ☐ The Myriad debugger, `moviDebug`, supports accessing many memories/registers by name rather than a specific address. For further details, see the debugger documentation (see section 15)
- ☐ The above table is an abridged copy of the one available in the Myriad Platform Design Databook (see section 15)

[redacted] Cache Hierarchy

8.3.1. List of System Caches

Processor	Type	Size	Associativity	Comment
SHAVE[N]	L1 Instruction	2 KB	2-way	N = {0,1,..11}
SHAVE[N]	L1 Data	1 KB	Directly mapped	N = {0,1,..11}
SHAVE[N]	L2	256 KB	2-way, 1-16 partitions	Shared instructions and data
LeonOS	L1 Instructions	32 KB	2-way	–
LeonOS	L1 Data	32 KB	2-way	Always write-through
LeonOS	L2	256 KB	4-way	Shared instructions and data
LeonRT	L1 Instructions	4 KB	2-way	–
LeonRT	L1 Data	4 KB	2-way	Always write-through
LeonRT	L2	32 KB	4-way	Shared instructions and data

8.3.2. Cache Usage Notes

- ☐ Ensure that caches are correctly initialized prior to use.
- ☐ Particular case of the above is that flushing the Leon L2 Caches prior to initialization/invalidation will yield unexpected results
- ☐ MDK provides a full set of APIs for cache manipulation which are recommended
- ☐ CMX address is `0x70------`, a non-cacheable alias is available at `0x78------`

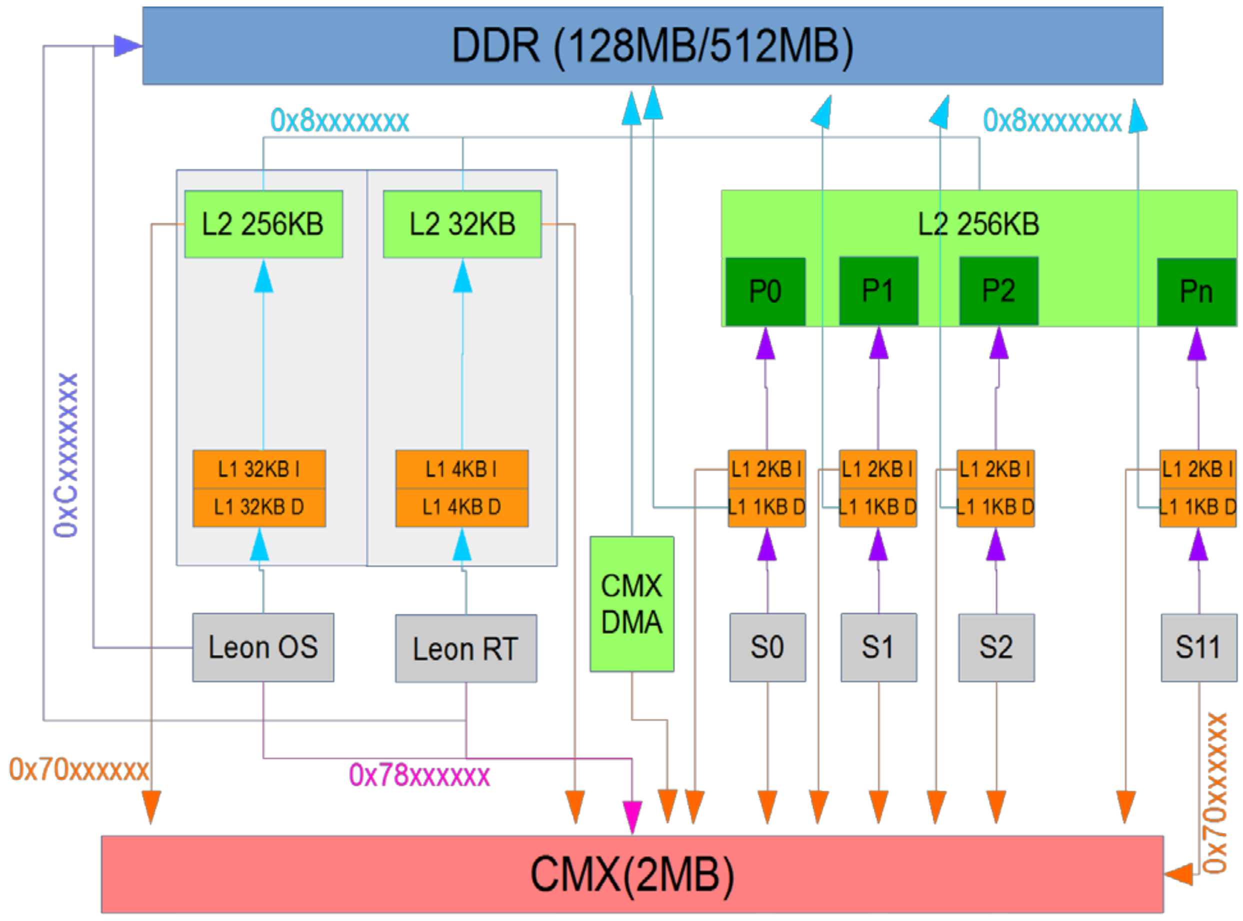

Figure 7: Architecture Diagram – The cache view

8.3.3. Cache Hierarchy Diagram

Fig. 7 above shows the Cache view architecture diagram. This is an important reference diagram to consider when using all of the processing cores of Myriad 2. Care must be taken in order to insure cache coherency between Leon OS, Leon RT and SHAVE cores if data is being shared between these. Each orange arrow in the above diagram represents an access that is done using 0x70xxxxxx type addresses and each cyan arrow represents 0x8xxxxxxx type addresses.CMX

8.3.4. Overview

The CMX acronym comes from Connection Matrix, which belies the fact it is comprised of several smaller SRAM blocks.

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

8.3.5. 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.

8.3.6. 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 two configurations supported as shown in Fig. 8.

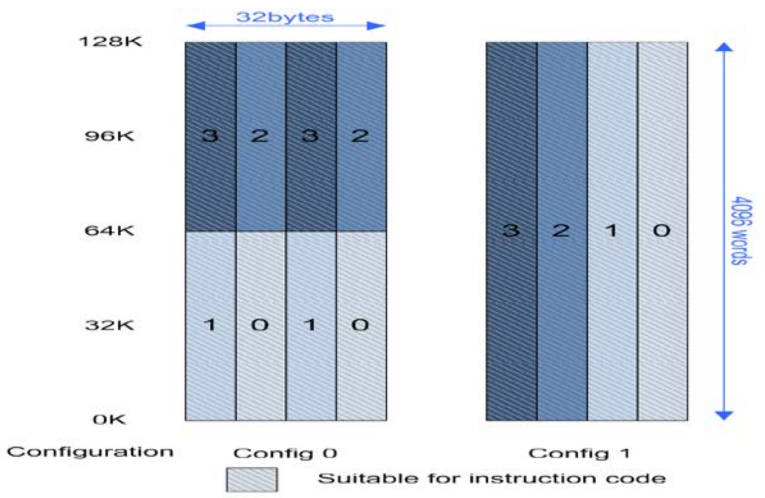

Figure 8: Supported CMX configurations

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.

[redacted] Data Layout Optimization

8.4.1. 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.

8.4.2. Sources & mitigations

8.4.2.1. 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 8.4.3) 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

8.4.2.2. 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

8.4.2.3. 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?

[redacted] Bandwidth

8.5.1. DDR Bandwidth

On MA2150 the part used is DDR2 533 MHz, giving a theoretical max throughput of 4 GB /sec. In practice between 60-70% of this is achievable.

8.5.2. 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.

9. MDK BUILD SYSTEM

[redacted] Introduction

This section 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.

[redacted] Overview of the build flow

9.2.1. Example diagram of building an application

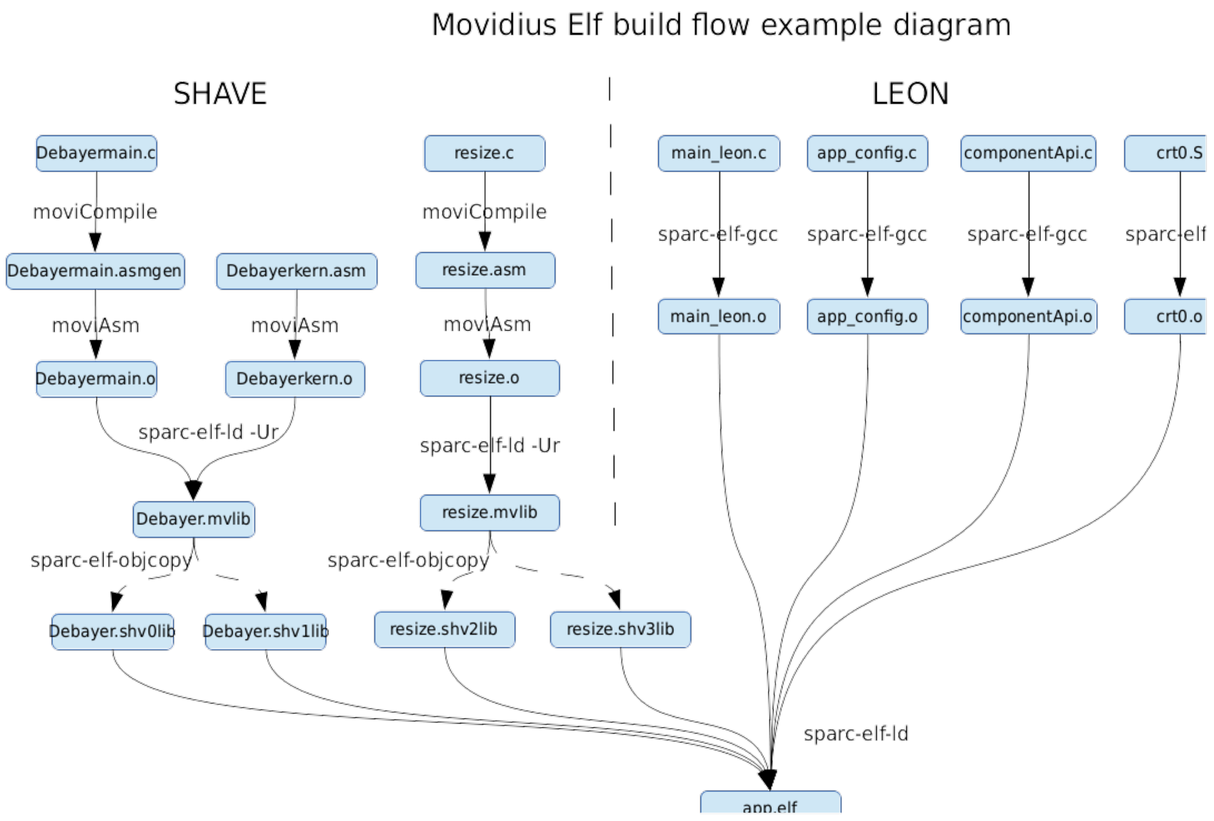

Figure 9: Application building example

9.2.2. 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.

9.2.3. 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 <u>ONE</u> 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.

Therefore, 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.

9.2.4. 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’). Therefore, 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.

1.1.2 Final steps

Next step is to map the SHAVE application/s onto the cores needed. In the example from Figure 9 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 5.2.2 General Concepts.

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 .

[redacted]Main build targets

9.3.1. 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 .

9.3.2. 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
- ☐ %.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, %.sh v11wndlib - 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`

This file contains the build rules for the SHAVE commonly used libraries. It 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 `includesettings.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

9.3.3. Mvlibs

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.

[redacted] Makefile: LEON code

9.4.1. 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.

9.4.2. 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.

9.4.3. 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
CCOPT        +=
```

9.4.4. Including components

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

```
# -------------------------[ Components used ]----------------------
--#
ComponentList = MvCV MvSTL CifGeneric Board 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
```

[redacted]Makefile: SHAVE code

9.5.1. 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 of course. 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 just listed the filter needed files and used a for each rule that may be blindly copied anywhere to generate a simple list of files

9.5.2. SHAVE application build rules

☐ Apps that do not 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 $@
```

[redacted] 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
```

9.6.1. 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
```

9.6.2. 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.

[redacted] 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.

9.7.1. make help

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

9.7.2. 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 ".

9.7.3. 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.

9.7.4. make start_server

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

9.7.5. make start_simulator

Starts moviSim in quiet mode for testing with the simulator.

9.7.6. 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.

9.7.7. make debugi

Starts moviDebug for interactive use.

9.7.8. 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.

9.7.9. 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.

9.7.10. make clean

Clean build files.

9.7.11. make load

Builds application and loads target elf into debugger but does not start execution (allows setting breakpoints in advance).

9.7.12. 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

9.7.13. make flash_erase

Erase SPI flash of the MV0xxx board.

Points to note:

- ☐ 'makeflash_erase' erases the flash. Recommended prior to interactive debugging

9.7.14. make show_tools

Displays tools paths.

[redacted] Multiple Myriad version support in MDK

9.8.1. Overview

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

- ☐ MA2100
- ☐ MA2150

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
    - ◦ MA2100
        - ▪ include – public header files for MA2100 (could be empty if there is no MA2100 specific public API extending the common one)
        - ▪ src – sources and internal headers for MA2100
    - ◦ MA2150
        - ▪ include – public header files for MA2150 (could be empty if there is no MA2100 specific public API extending the common one)
        - ▪ src – sources and internal headers for MA2150

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

9.8.2. 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. MA2100, MA2150.

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

9.8.3. Predefined macro

The build system automatically pre-define s a preprocessor macro with the capitalized string of Myriad 2 silicon version defined by the MV_SOC_REV variable, e.g. MA2100, MA2150.

10. RTEMS

[redacted] Overview

This section 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.

[redacted] 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

[redacted] RTEMS in MDK

RTEMS is shipped as prebuilt libraries in the Tools:

[tools dir]/[tools version]/common/rtems

RTEMS version: 4.10.99

[redacted] Writing RTEMS applications

Writing an RTEMS application is similar to 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
...
```

10.4.1. 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: Besides 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.

10.4.2. 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 */
```

[redacted] Working with threads

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

10.5.1. 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);
}
```

10.5.2. 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.

[redacted] Build system

10.6.1. Overview

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

Assumptions:

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

10.6.2. Requirements

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

- Autoconf 2.69
- GNU Make 4.0
- GCC 4.9.0
- Running in bash 4.3.18

10.6.3. First installation steps

- The following environment variable needs to be set:
`MV_RTEMS_SRC_PATH = /path/to/your/RTEMS/folder`
- Please go to `mdk/common/utils/RTEMSbuild`
- Please run “`make setup`”
- Because of various terminal/script settings the first setup cannot be run from inside a script. This step will print out three commands which must be run manually in the sequence they are printed.

This is only needed on the first run or after a "`make cleanall`" command. Once this is run the subsequent "`make all`" commands will work directly

- ☐ Optional: type "`make install`" if you wish to have all the libraries packed in a specific folder

10.6.4. Rebuilding

If there are any changes made to the RTEMS source files and an initial build was already started, all that is required to build in the changes is:

```
cd [path_to_mdk_installation]/mdk/common/utils/RTEMSbuild
make all
```

10.6.5. Using the new libraries

In order to use the newly build libraries in applications the following Makefile variable needs to be changed in the application Makefile.

For Leon OS this would be:

```
MV_RTEMS_LOS_LIB = $(MV_RTEMS_BUILD_DIR)/sparc-myriad-elf/myriad2/lib
```

For Leon RT this would be:

```
MV_RTEMS_LRT_LIB = $(MV_RTEMS_BUILD_DIR)/sparc-myriad-elf/myriad2/lib
```

where the MV_RTEMS_BUILD_DIR variable should be defined in your own project's Makefile pointing to the folder where RTEMS was built. By default this would be:

```
$MV_RTEMS_SRC_PATH/rtems-build
```

10.6.6. More details about customized builds

If desired, two different repositories may be kept for different configurations on each LEON. Building inside the source, independently of the MDK convenience file is done like:

```
cd [RTEMS source tree for desired processor]
./bootstrap -c
./bootstrap -p
./bootstrap
mkdir [build directory name]
cd [build directory name]
export PATH=$PATH:$MV_TOOLS_DIR/[desired tools version]/[linux64 | linux32 | win32]/sparc-myriad-elf-4.8.2/bin
../configure –-target=sparc-myriad-elf –-enable-drvmgr=no –-enable-posix –-enable-rtemsbsp=myriad2 –-enable-cxx –-disable-tests
make all
```

The configuration flags may be tweaked also but the "`--disable-tests`" flag should be kept as the default RTEMS testsuite does not compile directly for Myriad 2 causing a build fail if called.

[redacted]External references

[ref6] RTEMS Wiki
http://www.rtems.org/wiki/index.php/Main_Page

[ref7] RTEMS applications C user's guide
http://www.rtems.org/onlinedocs/doc-

current/share/rtems/html/c_user/index.html

[ref8] RTEMS POSIX API user's guide
http://www.rtems.org/onlinedocs/doc -
current/share/rtems/html/posix_users/index.html

[ref9] RTEMS POSIX 1003.1 compliance guide
http://www.rtems.org/onlinedocs/doc-
current/share/rtems/html/posix1003_1/index.html

[ref10] RTEMS BSP and device driver development guide
http://www.rtems.org/onlinedocs/doc-
current/share/rtems/html/bsp_howto/index.html

11. MDK SOFTWARE OVERVIEW

[redacted]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.

[redacted]Example Applications

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

[redacted]Including test data into an application

11.3.1. Overview

This section 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.

11.3.2. 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 will have:

```
extern u8 inputFrame;
```

[redacted] Intercommunication between processors

11.4.1. Overview

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

11.4.2. 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.

11.4.3. 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.

11.4.4. Using unified symbols

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

This may be achieved through symbols unifying.

Let us assume the previous myvar variable was declared in a SHAVE "mvlib" which was then loaded into both `shave3` and `shave4`. However, 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.

1.1.3 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_main;
```

This can then be used to take out the address of the starting point and pass it to `swcStartShave*` functions like for example:

```
swcStartShave(0, ( u32 )&someapp0_main);
```

The following code sequence shows the declarations needed and the sequence of function calls needed in order to start the execution of SHAVEs from the LEON side. Let us 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(entryPoints[i]);
      swcSetAbsoluteDefaultStack(entryPoints[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 SHAVE s to be used and extern u32 `appName0_entryPoint` is the SHAVE entry point function. Please refer to the section describing Makefile rules for applications for more details on how to use this.

11.4.5. 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
```

11.4.6. 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 9.5.2.

[redacted] Synchronous vs. Asynchronous SHAVE running functions

11.5.1. Overview

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

11.5.2. 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
```

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

11.5.3. 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));
```

[redacted]Dynamic Loading

11.6.1. Overview

This section aims to provide user understanding of how dynamically loading code to SHAVE cores for execution is possible.

11.6.2. File types that are dynamically loaded

Two types of executable files may be dynamically loaded:

- “Mbin” files
- “shvdlib” files

The first type involves loading executable files built from the deprecated “mof” file format. The second one uses “elf” built files. Dynamic loading is supported through the use of the `swcShaveLoader` library functions:

```
void swcLoadMbin ( u8 *sAddr, u32 targetS);
void swcLoadshvdlib ( u8 *sAddr, u32 targetS)
```

11.6.2.1. Mbin file type

In order to exemplify the usage of “mbin” files, the `SkypeUSBVideoCam` example will be used as a study material. The first thing to note is adding the “mbin” objects into the application's Makefile. Two raw data files are added: one made out of the “mbin” and one made out of the “mbin” symbols that hold symbolic information about “mbin”:

```
RAWDATAOBJECTFILES   =   $(SkypeUSBApp)  _mbin.o  $(DirAppOutput)
/usb.sym.o
```

Then, instead of configuring “mvlib” files we have an “mbin” configuration for SHAVE application to be built:

```
SHAVE_APP_LIBS = $(SkypeUSBApp) .mbin
```

Next, in the SHAVE local build rules, instead of having “mvlib” rules like for the rest of the examples, we have rules for generating the “mbin” file, the symbols file used for interfacing with the “mbin” and the object file that loads the “mbin” into memory:

```
#Describe the rule for building the SkypeUSB application. Simple rule
specifying
#how to build the required mbin and another rule showing how to link it
into a
#elf file to load in the final elf
$(SkypeUSBApp)        .mbin        :        $(SHAVE_SkypeUSB_OBJS)
$(PROJECT_SHAVE_MBINLIBS)
      $(ECHO)      $(MVLNK)      $(MVLNKOPT)                  -pad:16
$(PROJECT_SHAVE_MBINLIBS)    -mbin    $(SHAVE_SkypeUSB_OBJS)    -
symPrefix:SVE0_  -o:  $(SkypeUSBApp)  .mbin  -sym:  $(DirAppOutput)
/usb.sym.S

#keep right dependency in so that make works correctly and we can invoke
with -j
$(DirAppOutput) /usb.sym.S : $(SkypeUSBApp) .mbin

#We also need to list the rule for building the raw data object file to
keep the mbin information in
#we'll specify the address we want the raw data to stay at and the
symbol name we desire to use to address
#this data
$(SkypeUSBApp) _mbin.o : $(SkypeUSBApp) .mbin
      $(OBJCOPY) -I binary --reverse- bytes =4 --rename-section .
data  =.cmx.usbcore  --redefine-sym   _binary_ $(subst /,_,$(subst
.,_,$<)) _start =binary_usb_mbin_start -O elf32-sparc -B sparc $<
$@
```

The first rule is for building the “mbin”, the second rule sets dependencies correctly and the third rule takes the “mbin” file and stores it into a memory section(in this example in CMX). It also provides a “binary_usb_main_start” symbol so that we can pass that as parameter to the LEON function that loads “mbins” into SHAVEs.

Then, in the LEON code of the application:

```
/ Load USB mbin
swcLoadMbin(( unsigned char *)binary_usb_mbin_start, usbCoreSvu);
```

11.6.2.2. 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 ShaveHelloDynamic . There are two entry points: addshave_entry and subshave_entry , as there are two SHAVE applications, where “addshave” and “subshave” represent the appName , while “entry” represents the name of the entry point - “entry” 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 a “ adddyn ” or “ subdyn ” symbol so that we can pass that as parameter to the LEON function that loads “shvdlib” into SHAVE s.

Then, in the LEON code of the application:

```
//dynamically loadable code
extern u8 adddyn;
extern u8 subdyn;
swcLoadshvdlib(&adddyn, SHAVE_NR);
swcLoadshvdlib(&subdyn, SHAVE_NR);
```

[redacted] DDR configuration

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

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

11.7.1. 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 246 MHz (MA2100) / 264 MHz (MA215x)
- ☐ DDR is supplied directly from PLL1 bypassing the divider
- ☐ DDR frequency is 492 MHz (MA2100) / 528 MHz (MA215x)
- ☐ DDR is clocked on edges, so the data rate is 32 bit @ 492 MHz (MA2100) / 528 MHz (MA215x)

Since the on-the-fly reinitialization 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, then the RTEMS startup code takes care of not changing the PLL1, but it is important for the application not to set the DDR axillary 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, then 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 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,
   };
```

11.7.2. 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, application 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);
    ...
}
```

12. DRIVERS

[redacted] 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 Movidius as part of the MDK and RTEMS.

[redacted] RTEMS Drivers

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

12.2.1. 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.

12.2.1.1. Overview

The SD Driver provided in RTEMS is an 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 section is dedicated to explain the usage of this driver and its integration.

12.2.1.2. 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 and the device name

typedef struct {
        u8 slot;
        const char *devName;
} osDrvSdioEntry_t;

// Struct that keeps the number of card slots; Interrupt Priority that 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 operations, please go to:

https://docs.rtems.org/doxygen/cpukit/html/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 mounting 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/cpukit/html/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.

12.2.1.3. SD Driver Performance

For reference, Movidius has characterized the performance of different SD cards using FAT32 and 32 Kbyte clusters on a MV0182 running at 500 MHz. The data was obtained by writing 2 GB bytes of data in chunks of 20 MB bytes and reading them back for verification. SD cards were formatted using SDFormatter from SD Association under Windows.

SD Card	Average Write MB/s	Average Read MB/s
micro SDHC Maxwell, class 10, 8 GB	9.5	13
micro SDHC Maxwell, class 10, 8 GB	9.5	13
micro SDHC I SanDisk, class 10, 8 GB	11	13
micro SDHC Verbatim, class 10, 8 GB	10	13
micro SDHC I PNY, class 10, 16 GB	11	13
micro SDHC Toshiba, class 4, 4 GB	4	11.5
micro SDHC Apacer, class 6, 4 GB	6	13
SDHC I (3) ExtremePro SanDisk, class 10, 32 GB	14.5	13

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

12.2.1.4. 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 might be great. 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]
```

- End of document -