Freescale Semiconductor
Application Note

Document Number: AN4745
Rev 1, 06/2014

Optimizing Performance on Kinetis K-series MCUs

by: Melissa Hunter

Contents

1 Introduction

In embedded systems, resources are often limited and getting the best possible performance out of those resources can be critical. Although the idea of performance and low power might seem contradictory, the ability to perform a task quickly and then enter a low power mode can lead to an overall reduction in system power consumption. Therefore, almost any system can benefit from efforts to improve performance.

Increasing performance for an embedded system can be a complicated task. There are often nuances of the inner workings of the architecture and system features that impact the system. In addition, every system might have different performance goals. For example, some systems might be focused on pure CPU performance while another system might need to optimize throughput on a communications port like ethernet or USB.

This application note will explain the features found on Kinetis K-series devices that can affect system performance. The document is not a step-by-step guide on how to optimize an application as there isn't a hard set of rules that will work in all cases. The main goal is to explain the key architectural and system module features that can be tuned to optimize an application so that designers can make informed decisions when designing their system hardware and software.

2 Kinetis K-series architecture overview

The system architecture is one of the biggest factors in the overall system performance. How the different blocks fit together also has an impact on some of the module level features. So the first step to understanding how to optimize system performance is understanding the architecture from a high level.

The figure below shows a simplified block diagram of the Kinetis K70 family device. This family was selected because it shows the superset for the performance features that will be discussed. The other Kinetis family devices might not have all of the same features, but in general the overall architecture is largely the same.

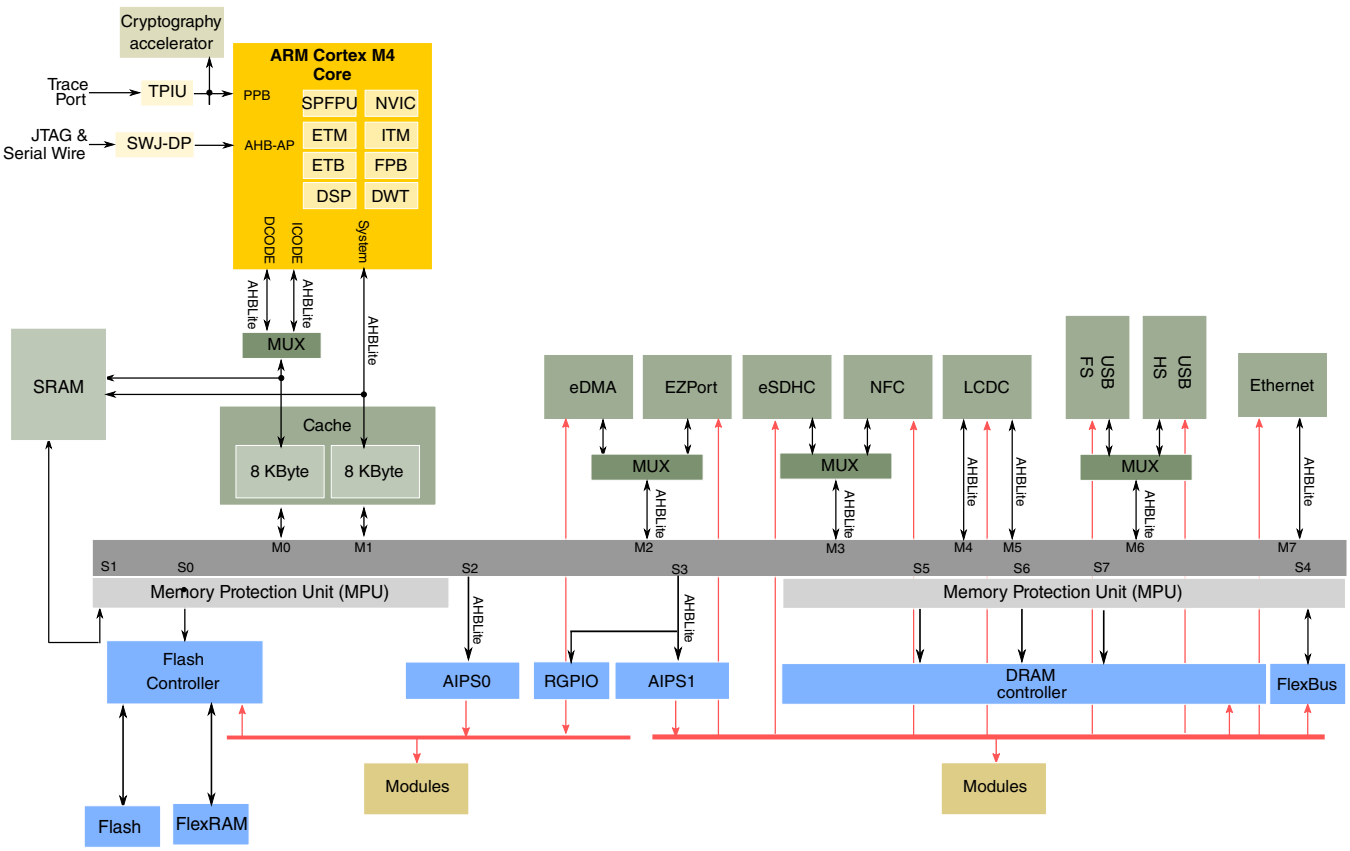

Figure 1. K70 block diagram

2.1 Core buses on Kinetis

The ARM Cortex M4 core uses a Harvard architecture with several memory mapped buses:

- ICODE - The ICODE bus is used for instruction accesses for any instructions stored between addresses 0x0000_0000-0x1FFF_FFFF.
- DCODE - The DCODE bus is used for data accesses for any instructions store between addresses 0x0000_0000-0x1FFF_FFFF.
- System - The system bus is used for all accesses to addresses between 0x2000_0000-0xDFFF_FFFF and 0xE010_0000-0xFFFF_FFFF.
- Private peripheral bus - The private peripheral bus (PPB) is mapped to addresses 0xE004_0000-0xE00FFFFF.

On Kinetis devices the ICODE and DCODE buses are multiplexed together to form a single CODE bus. Any core access below 0x2000_0000 will run on the CODE bus, and most accesses at 0x2000_0000 or above will run on the system bus (PPB accesses being the exception).

To an application, there isn't much distinction between the CODE and system buses; however, there is a difference in the performance of the two buses. CODE bus cycles have no delay added at the core. System bus cycle timing depends on the type of access. System bus data accesses have no delay added at the core, but instruction accesses add one wait state at the core.

2.2 Kinetis K-series memory map

To enable applications to use the CODE bus as much as possible, the Kinetis system memory map has been put together to include key memory regions at addresses below 0x2000_0000.

The table below shows the memory map regions for the CODE bus. These memory regions include aliased regions for the DRAM controller and FlexBus. Normally the DRAM and FlexBus regions are located on the system bus portion of the memory map. The aliased regions were added so that the memory is also available on the CODE bus. This allows for maximum performance when executing code from external memory. Whenever possible CODE bus regions should be used for storing code.

Table 1. CODE bus memory map

System 32-bit Address Range	Destination Slave	Access
0x0000_0000-0x07FF_FFFF	Program flash and read-only data	All masters
0x0800_0000-0x0FFF_FFFF	DRAM controller (aliased area) (optional)	Cortex-M4 core only
0x1000_0000-0x13FF_FFFF	FlexNVM (optional)	All masters
0x1400_0000-0x17FF_FFFF	FlexRAM/Programming acceleration RAM	All masters
0x1800_0000-0x1BFF_FFFF	FlexBus (aliased area) (optional)	Cortex-M4 core only
0x1C00_0000-0x1FFF_FFFF	SRAM_L: Lower SRAM	All masters

3 Kinetis SRAM

All Kinetis K-series devices include two blocks of on-chip SRAM. The first block (SRAM_L) is mapped to the CODE bus, and the second block (SRAM_U) is mapped to the system bus. The memory itself can be accessed in a single cycle, but because instruction accesses to the system bus incurs a one clock delay at the core, SRAM_U instruction accesses take at least two clocks.

SRAM_L is the only memory where code or data can be stored and the core is almost always guaranteed a single cycle access. For this reason, it makes sense to use the SRAM_L block as much as possible. This is a good area for storing critical code.

3.1 SRAM accesses

In addition to ports from the core for accesses on the CODE and system bus, the SRAM is also accessible to the non-core masters on the chip through a backdoor port. The backdoor port is one of the slave ports on the crossbar switch (XBS). There are three ports into the SRAM controller (CODE bus, system bus, and backdoor port) that map to accesses on one of the two SRAM blocks. The dual SRAM blocks allow the SRAM controller to handle simultaneous accesses to the SRAMs as long as those accesses are not to the same SRAM block. Allowable simultaneous accesses to the SRAM are:

- Core CODE (SRAM_L) and core system (SRAM_U) accesses
- Core CODE (SRAM_L) and non-core master to SRAM_U
- Core system (SRAM_U) and non-core master to SRAM_L

Strategic placement of code and data into each of the SRAM blocks can help to increase parallelism and overall performance. For a typical application, placing critical code in the SRAM_L block and placing data and the stack into the SRAM_U block will yield the best performance.

3.2 SRAM arbitration

Because the SRAM controller is dealing with accesses from more ports than there are SRAM blocks, the SRAM controller has its own internal arbitration logic. The arbitration is controlled by fields in the MCM_CR and allows for programmable arbitration modes for each of the two SRAM blocks.

The available SRAM arbitration modes are:

- Fixed CPU priority - In this mode the CPU always has the highest priority when accessing the corresponding SRAM block. This mode is recommended if single cycle access (or two cycle access for SRAM_U) is required.
- Fixed backdoor priority - In this mode the backdoor always has the highest priority when accessing the corresponding SRAM block. This mode might be used when trying to maximize bandwidth for one of the non-core masters; however, in some cases the special round robin mode might be preferred. See the special round robin mode description for more information.
- Round robin - In this mode priority switches between the core and the backdoor port trying to evenly distribute the priority between both ports. This mode is recommended for cases when a balance between CPU performance and non-core master bandwidth is needed. By default both SRAM blocks are configured for round robin arbitration.
- Special round robin - In this mode priority switches between the core and the backdoor port, but the algorithm favors the backdoor port. In many cases this might be an ideal setting for maximizing throughput for a non-core master. For example, if you are running an ethernet stack where buffers are stored in one of the SRAM blocks, the ENET module will need access to the SRAM, but the CPU is also required to process the packets that the ENET is sending or receiving. Giving the ENET priority might seem like the best setting to use, but could result in the ENET actually having to wait because the CPU hasn't had a chance to process packets. For such cases, it is good to experiment with the special round robin and fixed backdoor priority settings to determine which setting provides the best overall performance.

4 System cache

Some Kinetis devices include a system cache that can provide a significant increase to performance especially if executing code from external memory. Currently the system cache is only available on the 120/150 MHz Kinetis devices.

4.1 Cache organization and features

The system cache actually includes two separate cache blocks of 8 KB each. The first 8 KB cache is used for CODE bus accesses, and the second 8 KB cache is used for system bus accesses.

Cache controller features include:

- Two separate 8 KB cache arrays (16 KB total cache size)
- Two-way set associative cache structure
- 16-byte cache line size
- Supports write-back (copy-back), write-through, and non-cacheable modes
- 16 cache regions with independent cache modes

4.2 Cache regions and cache mode configuration

The cache mode settings are configured using pre-defined address regions. The cache controller supports up to 16 regions, but only ten are currently used on the Kinetis K-series devices. The region defines the address range to use and also a default cache configuration to use for that region.

The cache mode for a region can only be lowered from the initial value where:

```
write-back > write-through > non-cacheable
```

This means that the default cache setting for a region determines what cache modes are available. If a region is defined as non-cacheable by default, then the two cacheable settings are not available.

The table below shows the cache regions used on Kinetis including the default cache mode and available cache modes.

Table 2. Kinetis K-series cache regions

Region Number	Address Range	Destination Slave	Default Cache Mode	Available Cache Modes
R0	0x0000_0000 – 0x07FF_FFFF	Program flash and read-only data	Write-through	Write-through and non-cacheable
R1	0x0800_0000 – 0x0FFF_FFFF	DRAM Controller (Aliased Area)	Write-through	Write-through and non-cacheable
R2	0x1000_0000 – 0x17FF_FFFF	FlexNVM	Write-through	Write-through and non-cacheable
R3	0x1800_0000 – 0x1BFF_FFFF	FlexBus (Aliased Area)	Write-through	Write-through and non-cacheable
R4	0x1C00_0000 – 0x1FFF_FFFF	SRAM_L: Lower SRAM (ICODE/ DCODE)	Non-cacheable	Non-cacheable
R5	0x2000_0000 – 0x200F_FFFF	SRAM_U: Upper SRAM	Non-cacheable	Non-cacheable
R6	0x6000_0000 – 0x6FFF_FFFF	Flexbus (External memory - Writeback)	Write-back	Write-back, write-through, and non-cacheable
R7	0x7000_0000 – 0x7FFF_FFFF	DRAM Controller	Write-back	Write-back, write-through, and non-cacheable
R8	0x8000_0000 – 0x8FFF_FFFF	DRAM Controller - Write-through	Write-through	Write-through and non-cacheable
R9	0x9000_0000 – 0x9FFF_FFFF	FlexBus (External memory - Write-through)	Write-through	Write-through and non-cacheable

The flash regions default to write-through mode as that is the lowest possible cacheable mode. Writes to the flash do not use a memory mapped write bus cycle, so even though the flash regions are write-through, writes will not actually modify the flash and correct code should not be attempting write access to the flash over the CODE bus.

The SRAM regions are non-cacheable. This is because the cache does not accelerate SRAM accesses. A read from the SRAM block takes the same amount of time as a read from the cache (assuming a cache hit). There is no advantage to be gained from caching the SRAM (SRAM_L and CODE cache hits both take one clock cycle, while SRAM_U and system bus cache hits both take two clock cycles), so it is always non-cacheable.

4.3 Cache initialization

The system cache is disabled at reset. Here are the recommended steps for initializing the cache:

1. Modify the cache region configuration in the LMEM_PCCRMR from the default values if desired.
2. Set the LMEM_PCCCR[INVW1 and INVW0] bits to configure the controller to invalidate both ways of the CODE bus cache.
3. Set the LMEM_PCCCR[GO] bit to start the invalidate.
4. Wait for the LMEM_PCCCR[GO] bit to clear indicating the command has completed.
5. Enable the CODE bus cache by setting LMEM_PCCR[ENCACHE].
6. Set the LMEM_PSCCR[INVW1 and INVW0] bits to configure the controller to invalidate both ways of the system bus cache.
7. Set the LMEM_PCCCR[GO] bit to start the invalidate.
8. Wait for the LMEM_PSCCR[GO] bit to clear indicating the command has completed.
9. Enable the system bus cache by setting LMEM_PSCR[ENCACHE].

4.4 Cache coherency

The cache is only used for accesses originating from the core. There is no snooping capability, so accesses by non-core masters are not cached. If cached memory can be accessed by non-core masters, then cache coherency needs to be considered as part of the overall system design.

Here are some common cases where cache coherency should be considered:

Table 3. Cache coherency recommendations

Case	Recommendation Actions
Firmware will be used to erase and program flash space (this affects both the system cache and the FMC's cache and prefetch buffers)	Invalidate any cached lines from the flash in the CODE bus cache and also invalidate the FMC cache ways and prefetch buffer for the flash banks corresponding to the area that will be erased/programmed.
Non-core master (DMA, USB, ENET, SDHC, or NFC) will read/write a memory region	• If the non-core master will be accessing the memory within a known time frame where the core will not be accessing anything in or around the same area, then the relevant cached lines can be cleared before the non-core master is expected to read/write the memory. • If the non-core master will be accessing the memory within a known time frame, and the core will need access to memory in or around the same area, then the relevant cached lines can be cleared and then the cache disabled before the non-core master is expected to read/write the memory. Because the cache region setting cannot be changed from non-cacheable back to

Table continues on the next page...

Table 3. Cache coherency recommendations (continued)

Case	Recommendation Actions
	a cached mode, the entire cache (either CODE bus or system bus) would have to be disabled temporarily. • If the time when the non-core master will be accessing the memory is non-determinate, then the memory region should be configured for non-cacheable mode. This should be a rare situation. When the core and a non-core master are sharing data, there is usually some level of handshaking involved that would allow for the first two methods to be used. For example, the core configures a buffer for the ENET to receive data. The core would clear cached lines for the buffer location, then alert the ENET that the buffer is ready to receive. The ENET would notify the core when the buffer has been received and is ready to be read (either through interrupt or polling of a status bit). At which point, the core could read in the buffer loading the contents into cache as it reads.
Core is modifying data that will be read by a non-core master (LCDC)	• If using write-back mode, cache lines must be cleared before they will be available for the non-core master to read. • Use write-through mode for the memory region. Because the non-core master is only reading the data, the cache contents read by the core will never be stale. Write-through mode also means that the non-core master will have acccess to the latest data.

5 Flash Memory Controller (FMC)

For many systems the on-chip flash is the main memory. The flash memory controller (FMC) is the interface between the flash memory blocks and the system. In a typical configuration, the core and system bus clock speeds are clock significantly faster than the flash memory clock. The FMC includes features designed to accelerate flash accesses.

5.1 FMC features

The FMC has two key features that help to increase the chance that flash accesses can be serviced in a single clock cycle:

- FMC cache - There is a small cache within the FMC that stores recently accessed flash information. The exact configuration of the FMC cache can vary from device to device, but an FMC cache is present on all devices. Note: some Kinetis devices also contain a system cache that is completely separate from the FMC cache. The two caches operate independently, but can be used together to help accelerate flash reads.
- Prefetch speculation buffer - As memory accesses are usually sequential, when the FMC receives a request for a given flash location, the FMC will prefetch the next consecutive flash data chunk. Prefetched information is stored in the prefetch speculation buffer until a request to a different data chunk is received.

The FMC cache and prefetch speculation buffer allow the FMC to respond to flash accesses with no added wait states in many cases. Any time the requested information is available in the cache or prefetch buffer, the FMC responds with no added wait states.

5.2 FMC configuration

The FMC cache and prefetch buffers are enabled by default. Most applications will not require any reconfiguration of the FMC for optimal performance.

There are some programmable options that could be changed:

- Instruction vs. data cache - By default both instructions and data accesses are cached. This can be changed so that the entire FMC cache is used for instructions only or data only. The FMC cache could also be disabled entirely by turning off both instruction and data caching, but this setting is not recommended when trying to increase performance.
- Instruction vs. data prefetching - By default both instructions and data accesses can trigger a speculative prefetch cycle. This can be changed so that only instructions or only data accesses initiate a speculative prefetch. Instruction only prefetching might be desired if random data accesses are mixed in with mostly sequential instruction accesses to the same bank of flash.
- Cache locking - Each of the four ways in the FMC cache can be locked to force the cache to keep some values. The FMC cache is small, so usually it is a better option to move critical code or data to one of the SRAM blocks (preferrably SRAM_L) instead of locking the FMC cache. This way the critical information is available with no wait states and the entire FMC cache is still available for acceleration of flash accesses.
- Cache replacement control - The FMC cache replacement algorithm can be modified from the default setting where instruction and data are handled the same so that ways 0-1 or ways 0-2 are dedicated for instructions and remaining ways are used for data.

NOTE

The FMC registers should not be modified while accessing the flash. Freescale recommends executing any code that modifies the FMC settings from the on-chip SRAM.

6 Crossbar Switch (AXBS)

The AXBS is the primary bus interconnect for the microcontroller. The AXBS handles connections between the bus masters and the slave ports and also handles arbitration between masters when they are attempting to access the same slave.

NOTE

The Kinetis K-series 50 MHz devices do not support the features that are described in the following sections. Those devices use a crossbar switch lite (AXBS-Lite) with a reduced feature set. The AXBS-Lite configuration is fixed and cannot be modified to tune the operation specific to the needs of a given system.

6.1 AXBS accesses

One of the most important features of the AXBS is that it allows for simultaneous accesses from different masters as long as those masters are accessing different slave ports. Careful planning of the memory usage by masters in a system can create yield a significant increase in the overall system performance.

For example, here's a possible system memory configuration:

- Core - instructions in flash and core-only data and stack in SRAM_L
- USB - data buffers in SRAM_U
- LCD controller - graphic buffers in DDR

This memory configuration would allow all three of the masters to run the bus cycles it needs with very little interference from other masters. On occasion it would be expected that the core would need to access the USB buffers and make updates to the graphic buffers, but outside of these accesses the masters would be able to run in parallel.

6.2 AXBS arbitration

When two or more masters attempt to access a single slave port, the AXBS will use an arbitration algorithm to determine which master will get to access the port first. There are programmable fields that control the arbitration settings for each of the slave ports.

There are two different AXBS arbitration schemes that can be selected:

- Fixed priority - The master priorities are determined by the AXBS_PRSn register for the associated slave port. Use this setting if a particular master always needs to have the highest priority when accessing a given slave port.
- Round robin - In this mode each master's priority is based on the port's distance from the last master port that accessed the slave. For example, if master 2 was the last master to access a slave, then master 3 will have the highest priority for the next cycle while master 2 will have the lowest priority.

Keep in mind that arbitration only happens when there is more than one pending request to access a slave port. If a low priority master makes a request to an idle slave port, then the low priority master will get to start its bus cycle. If a higher priority master makes a request right after the low priority master started its bus cycle, the low priority master's bus cycle must reach a transfer boundary before the higher priority master gets its turn. For fixed length bursts the transfer boundary is at the end of the bus cycle.

6.3 AXBS arbitration during undefined length bursts

The AHBLite (officially AMBA AHB lite version 2.0) bus is the interface used to access the AXBS. Many of the transactions on the bus will be single transfers or fixed length bursts. The bus does have support for an undefined length burst bus cycle. Some masters like the ENET and USB-HS controller can request undefined length burst transfers.

The AXBS_MGPCRn[AULB] field determines which points during an undefined length burst transfer are treated as a bus boundary where arbitration can occur. This setting can be used to allow a higher priority master to gain mastership of a slave port in the middle of the burst instead of forcing it to wait for the entire burst to complete. The AULB can be configured for:

- No arbitration allowed during an undefined length burst
- Arbitration allowed between any beats of an undefined length burst
- Arbitration allowed after four beats of an undefined length burst
- Arbitration allowed after eight beats of an undefined length burst
- Arbitration allowed after 16 beats of an undefined length burst

6.4 AXBS parking

In addition to having options for arbitration, the AXBS also has the ability to park each slave port on a master. When the slave port goes idle, the AXBS will park it on the appropriate master. If the next access to the slave port is from the parked master, then the access starts immediately with no wait states added for the AXBS. If a port is idle and receives a request from a master that it is not parked on, then there is a one clock cycle delay while the AXBS switches to the master.

Like the arbitration, the parking settings are configured on a per slave basis configured by AXBS_CRSn[PCTL]. The parking options are:

- Fixed parking - In this mode the slave port will always park on the same master when the port is idle. The master to park on is controlled by the AXBS_CRSn[PARK] field. This setting would be used if there is only one master that will

ever access a given slave port, or if latency for a specific master needs to be guaranteed. By default all of the slave ports used the fixed parking option to park on master 0 (the Cortex M4 core).

- Park on last - In this mode the slave port will park on the last master that used the port. This setting would be used if most accesses to the slave port are expected to come in groups or if trying to share the port as evenly as possible between masters.
- No parking - In this mode the slave port is not parked on any master when the port goes idle. This means that any master would incur at least a one clock penalty when attempting to access the port. In general there are probably very few systems that would use the no parking option. This mode can be used to decrease power consumption within the AXBS module, but power savings would be small compared to overall power consumption in most cases.

7 Summary

- Know the application and priorities. Some optimizations will help increase overall performance, but many optimization options create a trade-off where performance is gained in one area and lost in other. Clear optimization goals are a must.
- Plan data movements and code location in advance. Not all memory addresses are created equal. Be aware of the latency involved in accessing different memory locations. Also, not all masters can access all addresses, and default cache modes vary (for devices with a cache).
- Use the SRAM_L and SRAM_U block for storing critical code and data. This is also a good location for the stack. The SRAM blocks are the fastest memory on the part so make sure to take full advantage of them.
- Take advantage of the flash acceleration features built into the flash memory controller (FMC). Even though it might be tempting to run the entire chip at 25 MHz to have a 1:1 clock ratio between the core and flash to eliminate wait states, unless there is an actual need for no wait state execution from the flash (possibly to meet determinism requirements), running the core at a higher frequency will still yield better overall performance. When running the core at a higher clock rate than the flash clock, there might be wait states in some cases, but the FMC is designed to minimize the occurrence of wait states.
- Use the system cache if one is available on your device. Cache hits are just as fast as storing code/data in the SRAM. Make sure to have cache management software in the system to avoid coherency issues.
- If external memory on the FlexBus or DDR controller will be used for code, make sure to use one of the aliased memory regions on the CODE bus for accessing instructions. This will save one clock cycle over accessing the memory through a system bus address. Using the system cache for these locations is also highly recommended.
- Use code optimizations wisely. Compilers will usually offer a choice of optimizing for speed or size. Optimizing for speed might seem like the best option for performance, but that is not always the case. If optimizing for sizes allows for mode code in the SRAM blocks or means that functions fit more easily in the cache, then performance might actually be best using size optimizations. Experiment with the switches that are available to find the optimal compiler settings.
- Parallelism is the best way to increase overall system performance. Take advantage of the crossbar switch (AXBS) and its ability to have concurrent, non-blocking transfers. The two SRAM blocks can also support concurrent accesses.
- When moving large blocks of data, use the DMA. The DMA can transfer data more efficiently than the core in many cases. Using the DMA will also free up the core to perform other tasks (more parallelism).
- Don't forget to look at the AXBS arbitration and parking settings. Some experimentation might be needed to find the best configuration.

8 Revision history

Table 4. Revision history

Revision number	Date	Substaitial changes
0	05/2013	Initial release
1	06/2014	Updated system bus wait state information

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4745
Rev. 1
06/2014