
Using L1 Cache on PIC32MZ Devices

This document provides information on the Level 1 (L1) CPU cache implementation in the PIC32MZ device family and describes the risks that are associated with a cached system. In addition, methods to address these risks are provided.

For advanced users, both the MPLAB® Harmony Integrated Software Framework cache management routines and the special instructions in the microAptiv™ core to manage the cache for optimum performance are also discussed.

Note: This document is not intended to be a comprehensive discussion of cache architectures or implementations, nor does it address every detail of cache management. For more information, refer to the microAptiv™ Microprocessor core software user manual, which is available by visiting: www.imgtec.com.

A familiarity of the PIC32MZ architecture is helpful to understand this document. **Section 50. “CPU for Devices with microAptiv™ Core”** (DS60001192), which is available from www.microchip.com, is suggested reading and will be a useful reference.

BACKGROUND

A CPU cache is a separate block of memory that is used to compensate for access time of main memory. A cache described as a Level 1 cache uses memory that is as fast as the CPU, so that as long as the CPU is accessing the cache, it will never have to wait for an instruction or data. Level 2 and Level 3 caches are used in conjunction with a Level 1 cache and have memory whose access times are greater than the CPU, but are less than main memory.

The PIC32MZ device family uses only a Level 1 cache. The L1 cache is divided into two parts, an instruction cache and a data cache. The CPU accesses to memory can be made directly to memory or through the cache.

Use of the cache is critical to achieving the maximum performance from the PIC32MZ device family. Memory accesses to and from the cache occur in a single clock cycle compared to several clock cycles when access occurs through the system bus.

PIC32MZ ARCHITECTURE

The following sections describe the L1 cache and two other segments of the PIC32MZ architecture, which are key to its operation and configuration.

Flash Prefetch Module

The Flash Prefetch module is used to hide Flash Wait states by fetching program Flash memory using a 128-bit data path, four times the width of the 32-bit CPU bus. It is predictive, assuming that the next needed data is the next line address. As long as the code does not branch, the next instruction is always available. A branch causes a stall while the new line is loaded. Registers in this module determine the Flash Wait states and enable prefetch operation. For more information on the Prefetch module refer to **Section 41. “Prefetch Module for Devices with L1 CPU Cache”** (DS60001183).

Memory Management Unit (MMU)

In all PIC32 architectures, CPU access to memory and peripherals is accomplished through virtual address space. The virtual address space is divided into five segments named KSEG0 through KSEG3 and KUSEG. Virtual KSEG0 and KSEG1 addresses are both mapped to the lower 512 MB of physical address space. Boot Flash, program Flash and data memory are accessible through both KSEG0 and KSEG1. Special Function Registers (SFRs) are accessed through KSEG1 only. KSEG1 is never cached. The cache policy of KSEG0 is programmable, and set during cache initialization in the start-up code.

The PIC32MZ architecture introduces a Translation Look-aside Buffer (TLB) based Memory Management Unit (MMU). The TLB can be configured to perform virtual to physical address translations for KUSEG, KSEG2, and KSEG3. The cacheability attributes of these segments are defined when configuring the TLB using the CPU Coprocessor 0 (CP0) EntryLo0 and EntryLo1 registers. Details regarding the Coprocessor 0 registers are described in **Section 50. “CPU for Devices with microAptiv™ Core”** (DS60001192).

Using L1 Cache on PIC32MZ Devices

L1 Instruction and Data Cache

As shown in [Figure 1](#), CPU accesses to system RAM can occur either directly or through the cache. Similarly, accesses to Flash can occur directly or through the cache. DMA access always occurs directly with RAM. All accesses except those between the CPU and cache involve the system bus.

At Reset, code execution occurs from KSEG1 using uncached access. This is necessary as the cache cannot be used until it is initialized by start-up code. Once the cache is initialized, code execution can occur from KSEG0, which utilizes the cache for optimum performance.

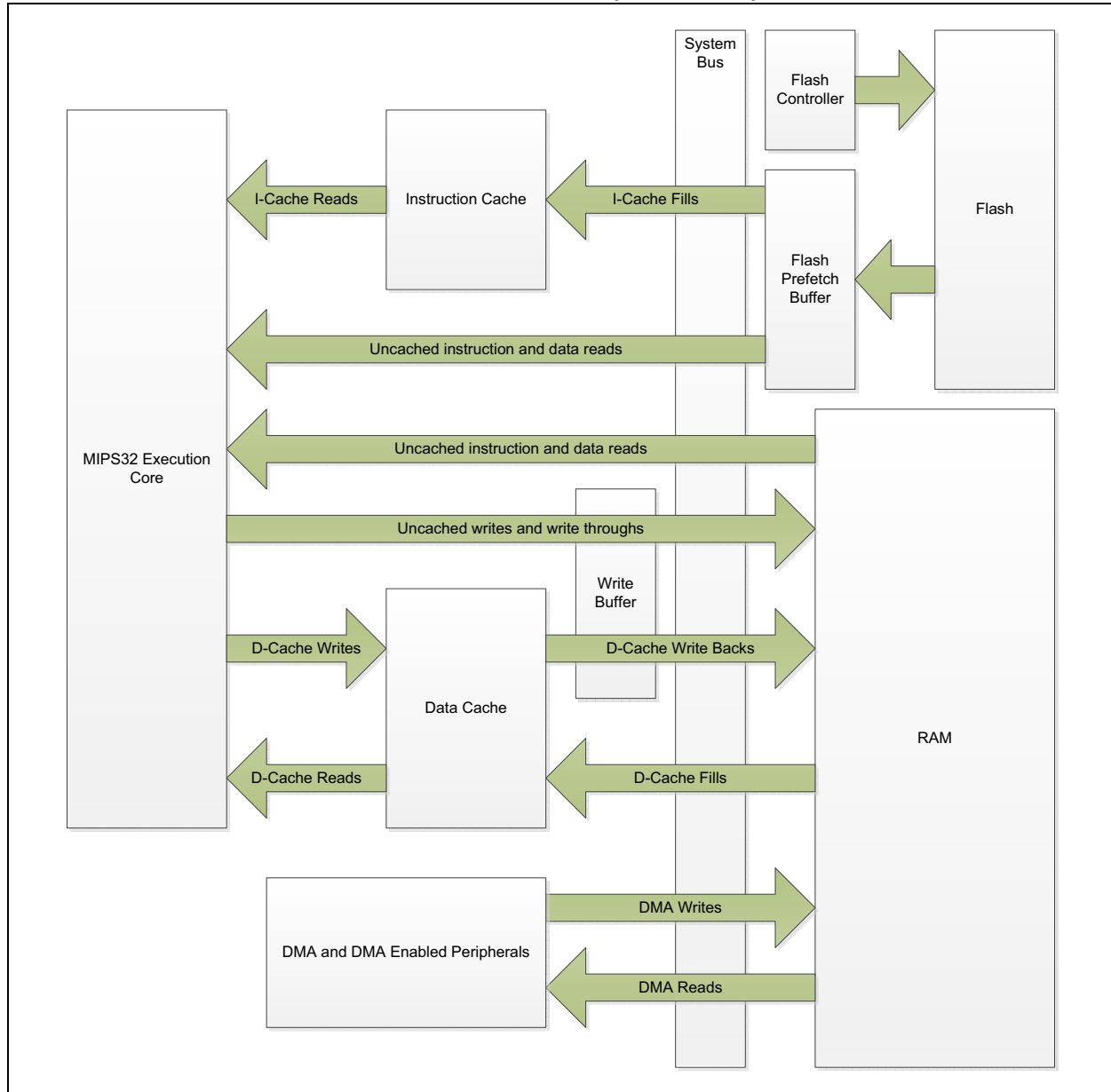
When the CPU fetches instructions or data from cached memory, the system checks if required data exists in the cache. If it exists, the data is read directly from the cache with no performance penalty. This is called a *cache hit*. If required data does not exist in the cache, hardware fills the cache with data from program Flash memory or data RAM, as required. The CPU must wait for the cache fill to complete before reading the data. This is called a *cache miss*. A cache miss incurs a performance penalty proportional to the time required to perform the cache fill.

The size of the cache is fixed at a fraction of the size of the program Flash or data RAM. The cache in the PIC32MZ is “four-way, set-associative”, it means the cache is divided into four equal parts (ways), distributed over the entire address range. Each way consists of a number of cache lines, which represent the smallest amount of data that can be transferred between the cache and the Flash or RAM. Any given memory location is mapped (associated) to a set location in the cache, and it can be mapped into one of the four ways within the set. Therefore, each memory location can be mapped into one of four locations within the cache.

Each cache line has an associated tag, which contains the memory mapping of the entry and the status bits. The PIC32MZ caches are virtually-indexed, physically tagged (VIPT), it means the cache line tags hold the physical address of the data in memory. The status bits identify the data in the cache line as valid or invalid, locked or unlocked and (for the data cache) dirty or clean. For more detailed information on cache organization, see **Section 50. “CPU for Devices with microAptiv™ Core”** (DS60001192).

On a cache miss, if no line allocated for that location in any of the four ways is available, one of the lines must be *evicted*. Hardware chooses the line to evict using a least recently used (LRU) algorithm. As the name implies, the cache line that was least recently used is evicted. Eviction frees the line in the cache and fills it with data from the new location. If the old data had been written to by the CPU, it is then written back to RAM before the location is filled with new data. This process is called a *cache write-back*. Cache data that has been written by the CPU and no longer matches that in RAM is called *dirty* and must be written back before it is replaced.

If the CPU were the only entity accessing memory, the cache would effectively be transparent to system operation. In a PIC32MZ device, DMA and DMA-enabled peripherals also access memory. When DMA writes to a location that is already loaded into the cache, the cache no longer represents the contents of RAM. When this occurs, the data in the cache is said to be *stale*. Similarly, locations that have been updated by the CPU for transfer to a DMA peripheral and are now dirty, must be written back to the cache before the DMA fetches the data.

FIGURE 1: PIC32MZ CACHE IMPLEMENTATION (SIMPLIFIED)

CACHE COHERENCY

Cache coherency refers to whether or not the data in the cache is synchronized with the contents of the physical memory to which it corresponds. Managing coherency involves maintaining synchronization when necessary, and recognizing those situations when synchronization is not needed.

There are many opportunities for the cache and memory to lose synchronization. This often happens when the CPU shares memory with another bus master. Some examples are described in the following sections. In these cases, coherency must be managed by software. Ignoring cache coherency can often lead to disastrous results, which can be hard to predict and difficult to troubleshoot.

Using L1 Cache on PIC32MZ Devices

DMA and Cache Coherency

Cache coherency must be considered in the system design when using DMA. This includes the DMA module and any other peripheral with built in DMA capability including the Flash Controller.

Ensure the following when using DMA:

- Data written by the CPU is available in RAM for the DMA peripheral when it needs it. It cannot be assumed that data written to the cache will have been written to RAM when the DMA reads memory, as the cache may still be dirty.
- CPU reads of memory updated by DMA are accurate. It cannot be assumed that the cache will contain the data that the DMA has written to RAM, as the cache may be stale.

Flash Updates and Cache Coherency

The PIC32MZ can reprogram Flash memory on-the-fly. Once the Flash has changed, any cached lines of the updated memory are now stale. Before the CPU accesses the updated Flash locations, the cache must be invalidated. The proper cache must be invalidated based on the use of the updated Flash region, i.e data cache for data and instruction cache if the Flash region contains executable code.

Executing Code from RAM and Cache Coherency

To execute code from RAM, the RAM must first be written. Writes to the RAM occur through the D-Cache while execution from RAM will occur through the I-Cache. Before the instruction is executed the D-Cache line containing the instruction must be written back to RAM and, if this results in an I-Cache line being stale, that line must be invalidated.

CACHE IMPLEMENTATION

Cache Policies

The cacheability attributes of KSEG0 are controlled by three bits in the CP0 Configuration register. The cache policy of the entire KSEG0 region is determined by these bits. For memory that is mapped into KUSEG, KSEG2 or KSEG3 using the TLB, the cache policy of each memory range can be specified individually in the TLB configuration registers.

The following four cache policies are supported:

- Uncached
- Cacheable, non-coherent, write-back, write allocate
- Cacheable, non-coherent, write-through, no write allocate
- Cacheable, non-coherent, write-through, write allocate

Cache policy descriptions are as follows:

- **Uncached:** Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- **Write-back with write allocation:** Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, but main memory is not written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. Therefore, the allocation policy on a cache miss is read-allocate or write-allocate. Data stores will update the appropriate dirty bit in the way-select array to indicate that the line contains modified data. When a line with dirty data is displaced from the cache, it is written back to memory.
- **Write-through with no write allocation:** Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, only main memory is written. Therefore, the allocation policy on a cache miss is read-allocate only.
- **Write-through with write allocation:** Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. In addition, the store data is also written to main memory. Hence, the allocation policy on a cache miss is read-allocate or write-allocate.

Appendix A: “Cache Policy and Coherency” demonstrates cache coherency and the effects of different cache policies on cache and physical memory contents after writes to virtual and physical memory using a simple program and MPLAB® X IDE.

Pros and Cons to Each Policy

[Table 1](#) summarizes some of the pros and cons for each cache policy.

The default cache policy for the PIC32MZ family, as is present in the start-up code supplied in the development tools, is write-back, write-allocate. For more information on changing the KSEG0 policy, refer to “*MPLAB® XC32 C/C++ Compiler User's Guide*” (DS51686). The start-up code also contains source code showing how the cache is initialized.

TABLE 1: CACHE POLICY COMPARISON

Policy	Pros	Cons
Uncached	No cache coherency issues.	Greatly impaired performance since every memory access must account for bus transfer time and memory wait states.
Write-back with Write Allocation	Best performance is achieved with this policy. All transactions are done using the cache with memory accesses performed only when needed.	Application must address coherency on both reads and writes to memory.
Write-through with no Write Allocation	Cache coherency issues are eliminated for writes as memory is always updated.	Results in the CPU taking a larger percentage of the memory bus bandwidth since every CPU write results in a bus transaction. Even back to back writes are written to memory. Cache coherency for CPU reads must still be addressed.
Write-through with Write Allocation	Cache coherency issues are eliminated for writes as memory is always updated. Writes to memory also fill cache so the data written is immediately available for a CPU read.	Results in the CPU taking a larger percentage of the memory bus bandwidth since every CPU write results in a bus transaction. Even back to back writes are written to memory. Cache coherency for CPU reads must still be addressed. Writes to memory also fill the cache which can result in needed data being evicted from cache.

KSEG0 and KSEG1

In MIPS cores, the first 512 MB of physical memory can be either cached or uncached, depending on the virtual address used. This 512 MB region is referred to as KSEG0 when cached, and KSEG1 when uncached. KSEG0 addresses are in the range of 0x80000000 through 0x9FFFFFFF. KSEG1 addresses are in the range of 0xA0000000 through 0xBFFFFFFF. With the exception of the Special Function Registers, which exist only in KSEG1, KSEG0 and KSEG1 both point to the same physical memory. Accesses using KSEG0 are cacheable while accesses through KSEG1 are uncached. The cacheability attribute of KSEG0 can be set using the CPU Coprocessor 0 (CP0) Configuration register. By default it is set to write-back, read-allocate.

In PIC32MZ devices, the linker allocates data sections to cached KSEG0 segment as specified in the linker script. The MPLAB development tools provide functions for run-time heap allocation and link-time data variable allocation.

Link-time coherent allocation in KSEG1 is accomplished using the `coherent` variable attribute. The code in [Example 1](#) will create a 1024 element unsigned integer array, which will be allocated in KSEG1.

Since the default stack is allocated to the cached KSEG0 region, run-time allocation of uncached memory must come from the heap. The development tools provide two functional equivalents to `malloc` and `free` to create uncached variables at run-time. These functions are `pic32_alloc_coherent` and `pic32_free_coherent`. [Example 2](#) shows a function that allocates a 1024 element character buffer, utilizes it, and then returns it to the heap when finished.

Using L1 Cache on PIC32MZ Devices

EXAMPLE 1: USING THE LINK-TIME COHERENT ATTRIBUTE

```
unsigned int __attribute__((coherent)) buffer[1024];
```

EXAMPLE 2: ALLOCATING/FREEING COHERENT MEMORY AT RUN-TIME

```
#include <xc.h>

void myFunction(void) {
    char* buffer = __pic32_alloc_coherent(1024);
    if (buffer) {
        /* do something */
    }
    else {
        /* handle error */
    }
    if (buffer) {
        __pic32_free_coherent(buffer);
    }
}
```

TABLE 2: ADDRESS TRANSLATION MACROS

Macro Name	Description
KVA_TO_PA(v)	Translate a kernel (KSEG) virtual address to a physical address.
PA_TO_KVA0(pa)	Translate a physical address to a KSEG0 virtual address.
PA_TO_KVA1(pa)	Translate a physical address to a KSEG1 virtual address.
KVA0_TO_KVA1(v)	Translate a KSEG0 virtual address to a KSEG1 virtual address.
KVA1_TO_KVA0(v)	Translate a KSEG1 virtual address to a KSEG0 virtual address.
IS_KVA(v)	Evaluates to 1 if the address is a kernel segment virtual address, zero otherwise.
IS_KVA0(v)	Evaluate to 1 if the address is a KSEG0 virtual address, zero otherwise.
IS_KVA1(v)	Evaluate to 1 if the address is a KSEG1 virtual address, zero otherwise.
IS_KVA01(v)	Evaluate to 1 if the address is either a KSEG0 or a KSEG1 virtual address, zero otherwise.

[Example 3](#) shows a method for accessing a variable defined for KSEG0 from KSEG1.

EXAMPLE 3: CODE EXAMPLE FOR ADDRESS TRANSLATION MACRO

```
/* Declare a variable. By default it is defined in KSEG0 */
int Var1InKseg0 = 5;

/* Declare a pointer to the same variable type and assign */
/* it the translated address using the address translation */
/* macro */
int *pVar1InKseg1 = KVA0_TO_KVA1(&Var1InKseg0);

/* Assign x the value of the variable from the cache */
x = Var1InKseg0;
/* Assign y the uncached value */
y = *pVar1InKseg1;
```


PIC32MZ Cache Management Instructions

The Imagination Technologies Limited MIPS core includes special assembly instructions for managing the cache. Each instruction operates on a single cache line. Please refer to the Imagination Technologies Limited website (www.imgtec.com) for details regarding the use and operation of these instructions.

CACHE INSTRUCTION

The `CACHE` instruction is primarily used at start-up to initialize the cache. To manage coherency, the `CACHE` instruction can be used to:

- Invalidate a Cache Address Hit – Searches the cache for the specified address, and if a hit occurs, invalidates (evicts) the cache line. No write back is performed even if the cache line is dirty.
- Write back a Cache Address Hit – Searches the cache for the specified address, and if a hit occurs and the line is dirty, writes the cache line to memory.
- Fill Cache – Fills the instruction cache (I-Cache) with data from the specified address. For data cache (D-Cache) fills the `PREF` instruction is used.
- Fetch and Lock – Fills the instruction or data cache with data from the specified address, and locks it in the cache. The data remains locked in the cache until it is invalidated with the cache instruction.

PREF INSTRUCTION

The Prefetch instruction (`PREF`) is provided to optimize D-Cache performance by allowing software to specify the optimum cache fill operation:

- Fill a cache line for write operation. A cache line is reserved but no fill from memory is performed as memory will ultimately be written with new data.
- Fill a cache line for a read operation. A cache line is reserved and filled with the contents of memory.
- Streamed and Retained Options allow the user to specify a cache hierarchy where streamed data will not evict retained data.
- Write back and invalidate allows the user to free a cache line, writing it back to memory if it is dirty. Sometimes referred to as 'nudge'.

SYNCI INSTRUCTION

When initializing memory to execute from RAM, both the D-Cache and I-Caches are involved. This instruction is used when programming RAM with code to execute by synchronizing caches to make an instruction write effective. `SYNCI` forces a D-Cache write back and I-Cache invalidate on a specific address.

Refer to “*MPLAB® XC32 C/C++ Compiler User's Guide*” (DS51686) and the related ReadMe file for details regarding macro support for these instructions.

MPLAB Harmony Cache Management Functions

The MPLAB Harmony Integrated Software Framework includes several functions to help make cache management easier. These functions use the MIPS assembly instructions to perform operations on the cache as a whole, or over a range of addresses. These functions are part of the Device Control (DEVCON) System Service Library. Use of these functions is the recommended method of managing the cache.

The instructions allow the programmer to perform the following cache operations at run-time:

- Initialization
- Flush (entire cache, data, instruction or both)
- Clean (address range, data or instruction)
- Invalidate (address range, data or instruction)
- Lock (address range, data or instruction)
- Sync
- KSEG0 Policy Set or Read

Please see the latest MPLAB Harmony documentation for a complete description of these functions, including sample applications demonstrating their use.

Using L1 Cache on PIC32MZ Devices

METHODS OF ADDRESSING CACHE COHERENCY ISSUES

This section describes the available options for addressing cache coherency issues.

DMA

DMA peripherals always access physical memory and cache coherency must be addressed if the cache is enabled. The user has the option of using the default write-back write-allocate policy or changing to a write-through policy (with write allocation turned on or off). The recommended approach is to start with a write-back write-allocate (default) cache policy and use

KSEG1 when accessing any memory used by a DMA peripheral. This is the simplest approach and in most cases it will provide acceptable performance. Once the project is running and debugged, performance can be improved by changing the access of DMA memory to KSEG0 and employing the `CACHE` and `PREF` instructions to manage coherency. In systems employing multiple DMA bus masters, software management of the cache can be used only where necessary and implemented on one DMA peripheral at a time simplifying the debug process.

[Table 3](#) lists the DMA cache coherency management options.

TABLE 3: DMA CACHE COHERENCY MANAGEMENT OPTIONS

Method	Description	When to Use
Write-Back Write-Allocate Cache Policy using KSEG1 to Access DMA Shared Memory	Cache policy is set to write-back write-allocate but all DMA shared memory is referenced by the CPU using uncached KSEG1. This is the simplest cache management approach.	This is the recommended approach to get your project up and running.
Write-Back Write-Allocate Cache Policy with Software Managing Coherency	Cache policy is set to write-back write-allocate and DMA shared memory is referenced by the CPU using cached KSEG0. Cache coherency is managed using the cache management functions provided by the MPLAB Harmony Integrated Software Framework, invalidating cache lines and writing back data as needed. This method provides the best performance but requires software to manage the cache.	Use this approach to achieve the best performance when needed.
Software Management with a Write-Through Cache Policy	Cache policy is set using a write-through mode. DMA shared memory can be referenced using KSEG0 and no cache coherency issues will exist for writes. Coherency for reads must be managed using the <code>CACHE</code> or <code>PREF</code> instructions or directing all reads through KSEG1. The write-through policy eliminates the need to write-back cache data at the expense of increased system bus bandwidth utilization.	Use this approach to achieve the best performance when needed.

Updates of Flash

Flash updates can occur when a portion of Flash is used for non-volatile storage or in boot loader applications.

FLASH PROGRAMMING

When using Row Programming, the Flash controller can utilize DMA to read the contents of RAM and program Flash memory using the following methods:

- Initialize ROW data using uncached KSEG1
- Initialize ROW data using cached KSEG0 and use the `CACHE` write-back instruction or MPLAB Harmony cache management routines to force the cached contents to be written to RAM prior to initiating the NVM Write command

NON-VOLATILE DATA STORAGE

In non-volatile data storage Flash applications, there is no problem for write-back since writing of Flash occurs only through the Flash Controller. Cached lines will become stale once the Flash locations are reprogrammed. The following options are available:

- Access non-volatile data storage through KSEG1. If access to these locations is infrequent, this may be the best solution since it will not consume excessive bus bandwidth.
- Access using KSEG0 using any cache policy (except turning the cache off of course) being sure to invalidate cache locations corresponding to Flash memory locations that have been reprogrammed

BOOTLOADING

Bootloading, a process that updates the application code, is typically run infrequently and involves a large amount of Flash memory, which could potentially span the entire cache. Because of these characteristics, the best solution is to invalidate the entire I-Cache before executing any bootloaded code. If the Bootloader performs a reset to start execution of the new code, the start-up code will initialize the cache and no specific action is needed. However, if a reset is not performed, it is recommended that the cache be invalidated prior to executing the new code.

Executing Code from RAM

When loading RAM with data that will later be executed as code, both the I-Cache and D-Cache are involved. The `SYNCHI` instruction is provided specifically to solve this problem. After the executable data is written to RAM and before it is executed the `SYNCHI` instruction is used to synchronize both caches.

Data Organization

Caches perform best when spatial locality is employed in the data. As previously mentioned, the smallest amount of data that can be transferred to or from the cache is a cache line.

In PIC32MZ devices, a cache line is four words (16 bytes). Performance is optimum when static data that spans multiple words is declared so that is aligned to the cache line. This prevents a four word data structure from spanning two cache lines taking twice as long to load as a single cache line. Forcing groups of static single word variables that are used together into the fewest number of cache lines will improve performance. One method of doing this is to declare them in a structure that is aligned to a cache line.

CONCLUSION

Optimum performance using the PIC32MZ family of devices can only be achieved when the L1 CPU cache is used. Anytime DMA is used, the developer must plan for cache management in the design of the project to avoid coherency issues. Options that exist for managing cache coherency, range from simple to complex based on the desired system performance.

Using L1 Cache on PIC32MZ Devices

APPENDIX A: CACHE POLICY AND COHERENCY

The following four examples use MPLAB X IDE to help illustrate the effects of cache policy on cache coherency when the data memory is modified through KSEG0 and KSEG1.

- [Example 1: Uncached Writes to Data Memory](#)
- [Example 2: Write-back, Write-allocate Writes to Data Memory](#)
- [Example 3: Write-through, Write-allocate Writes to Data Memory](#)
- [Example 4: Write-through, No-write-allocate Writes to Data Memory](#)

For simplicity, these examples use KSEG0 and KSEG1 pointers to the address of the same memory location in SRAM. The KSEG0 pointer represents a cached CPU memory access. The KSEG1 pointer represents an uncached memory access. Even though the KSEG1 memory access is performed by the CPU, the cache has no knowledge of it, so it is equivalent to a DMA access to physical memory.

A.1 Example 1: Uncached Writes to Data Memory

This example shows two different data writes to the same memory location, the first using a KSEG0 pointer and the second using a KSEG1 pointer. The value is then read back through the same pointers into two different global variables. The cache policy is “uncached”. As expected, both reads yield the same value, equal to the second write (Figure A-1).

FIGURE A-1:

```

68  volatile unsigned int ks0;
69  volatile unsigned int ks1;
70
71  #define UNCACHED 0x02
72  #define WB_WA    0x03
73  #define WT_WA    0x01
74  #define WT_NWA   0x00
75
76  void set_cache_policy(int cc)
77  {
78      unsigned int cp0;
79
80      cp0 = _mfc0(16, 0);
81      cp0 &= ~0x03;
82      cp0 |= cc;
83      _mtc0(16, 0, cp0);
84  }
85
86  int main(int argc, char** argv) {
87
88      set_cache_policy(UNCACHED);
89
90      *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91      *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93      ks0 = *(volatile unsigned int*)0x80000300;
94      ks1 = *(volatile unsigned int*)0xA0000300;
95
96      while(1);
97
98      return (EXIT_SUCCESS);
99  }

```

Search Results		Output		Tasks	U
Name	Type	Address	Value		
ks0	unsigned int	0x800002A8	0xF005BA11		
ks1	unsigned int	0x800002A4	0xF005BA11		

Using L1 Cache on PIC32MZ Devices

As shown in [Figure A-2](#) and [Figure A-3](#), when we look at the virtual and physical memory using MPLAB X IDE, we can see that they both contain the same correct value.

FIGURE A-2:

Search Results		Output				Tasks
	Address	00	04	08	0C	ASCII
	8000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
	8000_0300	F005BA11	D498BD89	7B2C88E0	C772DF9F ,{..r.
	8000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
	8000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x...- P.z.yh..
	8000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
	8000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
	8000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H..2.E ..].M.>.
	8000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..0.."..Q

Memory Format

FIGURE A-3:

Search Results		Output				Tasks
	Address	00	04	08	0C	ASCII
	A000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
	A000_0300	F005BA11	D498BD89	7B2C88E0	C772DF9F ,{..r.
	A000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
	A000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x...- P.z.yh..
	A000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
	A000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
	A000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H..2.E ..].M.>.
	A000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..0.."..Q

Memory Format

A.2 Example 2: Write-back, Write-allocate Writes to Data Memory

Using the “write-back, write-allocate” cache policy, the CPU writes to the cache when using a KSEG0 address, and the data is not written back to SRAM until the line is evicted. When using a KSEG1 address, the CPU writes directly to SRAM, bypassing the cache. Therefore, we would expect to see the first data write in cache, but the second in the SRAM itself (Figure A-4).

FIGURE A-4:

The screenshot shows a code editor with the following C code:

```

68 volatile unsigned int ks0;
69 volatile unsigned int ks1;
70
71 #define UNCACHED 0x02
72 #define WB_WA    0x03
73 #define WT_WA    0x01
74 #define WT_NWA   0x00
75
76 void set_cache_policy(int cc)
77 {
78     unsigned int cp0;
79
80     cp0 = _mfc0(16, 0);
81     cp0 &= ~0x03;
82     cp0 |= cc;
83     _mtc0(16, 0, cp0);
84 }
85
86 int main(int argc, char** argv) {
87
88     set_cache_policy(WB_WA);
89
90     *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91     *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93     ks0 = *(volatile unsigned int*)0x80000300;
94     ks1 = *(volatile unsigned int*)0xA0000300;
95
96     while(1);
97
98     return (EXIT_SUCCESS);
99 }
100

```

Below the code editor is a memory dump table:

Search Results		Output		Tasks		Usage	
Name	Type	Address	Value				
ks0	unsigned int	0x800002A8	0xDEADBEEF				
ks1	unsigned int	0x800002A4	0xF005BA11				

Using L1 Cache on PIC32MZ Devices

Looking at the virtual versus physical memory in the MPLAB X IDE watch window (Figure A-5 and Figure A-6), we see that the same memory location shows different data, because the cache is now non-coherent. The cache will remain non-coherent (for this line) until the data in the cache is written back to SRAM.

FIGURE A-5:

Search Results		Output				Tasks
	Address	00	04	08	0C	ASCII
	8000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
	8000_0300	DEADBEEF	D498BD89	7B2C88E0	C772DF9F ,{..r.
	8000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
	8000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x...-. P.z.yh..
	8000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
	8000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
	8000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H..2.E ..].M.>.
	8000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..0.."..Q

Memory Format

FIGURE A-6:

Search Results		Output				Tasks
	Address	00	04	08	0C	ASCII
	A000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
	A000_0300	F005BA11	D498BD89	7B2C88E0	C772DF9F ,{..r.
	A000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
	A000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x...-. P.z.yh..
	A000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
	A000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
	A000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H..2.E ..].M.>.
	A000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..0.."..Q

Memory Format

As shown in Figure A-7 through Figure A-9, note that if we eliminate the KSEG0 write, the cache is never written to, so when we do the KSEG0 read, the data is fetched from physical memory, and the cache is updated at that time (read-allocate).

FIGURE A-7:

```

68 volatile unsigned int ks0;
69 volatile unsigned int ks1;
70
71 #define UNCACHED 0x02
72 #define WB_WA    0x03
73 #define WT_WA    0x01
74 #define WT_NWA   0x00
75
76 void set_cache_policy(int cc)
77 {
78     unsigned int cp0;
79
80     cp0 = _mfc0(16, 0);
81     cp0 &= ~0x03;
82     cp0 |= cc;
83     _mtc0(16, 0, cp0);
84 }
85
86 int main(int argc, char** argv) {
87
88     set_cache_policy(WB_WA);
89
90     /*(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91     *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93     ks0 = *(volatile unsigned int*)0x80000300;
94     ks1 = *(volatile unsigned int*)0xA0000300;
95
96     while(1);
97
98     return (EXIT_SUCCESS);
99 }

```

Search Results		Output		Tasks		Us
Name	Type	Address	Value			
ks0	unsigned int	0x800002A8	0xF005BA11			
ks1	unsigned int	0x800002A4	0xF005BA11			

Using L1 Cache on PIC32MZ Devices

FIGURE A-8:

Search Results

Output

Tasks

Address	00	04	08	0C	ASCII
8000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
8000_0300	F005BA11	D498BD89	7B2C88E0	C772DF9F,{...r.
8000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
8000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x..-. P.z.yh..
8000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
8000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
8000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H...2.E ..].M.>.
8000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..□.."Q






Memory

Data Memory

Format

Data

FIGURE A-9:

Search Results			Output			Tasks
	Address	00	04	08	0C	ASCII
	A000_02F0	AB969101	BB40DFF9	BE6A0858	EA20F6D0@. X.j... .
	A000_0300	F005BA11	D498BD89	7B2C88E0	C772DF9F,{...r.
	A000_0310	DFAA5032	105313A0	3C16A311	A380347F	2P....S. ...<04..
	A000_0320	78D7A432	052DA7EA	BE7A8950	D1A26879	2..x..-. P.z.yh..
	A000_0330	6E682C43	72CC9CFF	48695403	5A6E07FD	C,hn...r .TiH..nZ
	A000_0340	EFDD0909	77C06949	E7D18617	3F55B81AIi.wU?
	A000_0350	E048C411	45D132C8	EE5D08C6	D43EBE4D	..H...2.E ..].M.>.
	A000_0360	5A390006	4F84B36A	D97FBB95	5113221B	..9Zj..O ..0.."Q
Memory Data Memory Format Data						

Note that the cache will read-allocate in all modes except “uncached”. This can have important consequences when debugging, as the act of running a debug executive or reading data to print/display can make the difference between cache coherency and non-coherency.

A.3 Example 3: Write-through, Write-allocate Writes to Data Memory

Write-through with write allocation works similar to the write-back with write-allocation; the difference being that the data written to cache is also written to physical memory. We can illustrate this by eliminating the KSEG1 write, and verifying that both the KSEG0 read and KSEG1 read return the same data (Figure A-10).

FIGURE A-10:

The screenshot shows a code editor window titled 'main.c' with the following C code:

```

68 volatile unsigned int ks0;
69 volatile unsigned int ks1;
70
71 #define UNCACHED 0x02
72 #define WB_WA    0x03
73 #define WT_WA    0x01
74 #define WT_NWA   0x00
75
76 void set_cache_policy(int cc)
77 {
78     unsigned int cp0;
79
80     cp0 = _mfc0(16, 0);
81     cp0 &= ~0x03;
82     cp0 |= cc;
83     _mtc0(16, 0, cp0);
84 }
85
86 int main(int argc, char** argv) {
87
88     set_cache_policy(WT_WA);
89
90     *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91     /*(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93     ks0 = *(volatile unsigned int*)0x80000300;
94     ks1 = *(volatile unsigned int*)0xA0000300;
95
96     while(1);
97
98     return (EXIT_SUCCESS);
99 }

```

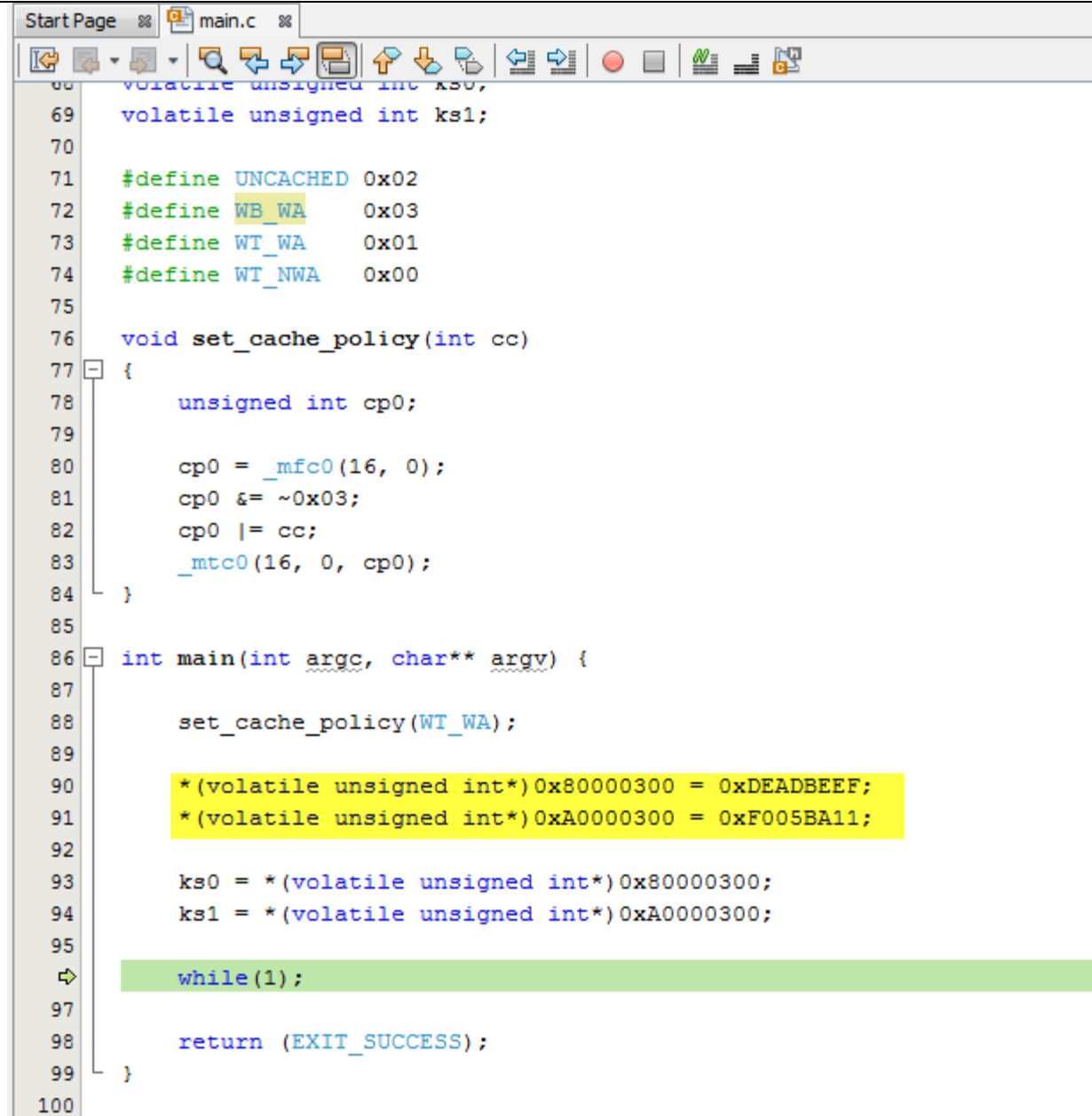
Below the code editor is a table with four columns: Search Results, Output, Tasks, and Us. The table contains two rows of data:

Search Results	Output	Tasks	Us
Name	Type	Address	Value
ks0	unsigned int	0x800002A8	0xDEADBEEF
ks1	unsigned int	0x800002A4	0xDEADBEEF

Using L1 Cache on PIC32MZ Devices

As shown in Figure A-11, if we do both writes, however, we are non-coherent again. The cache never saw the write to KSEG1, so it still thinks the first write is valid.

FIGURE A-11:



```
68 volatile unsigned int ks0;
69 volatile unsigned int ks1;
70
71 #define UNCACHED 0x02
72 #define WB_WA 0x03
73 #define WT_WA 0x01
74 #define WT_NWA 0x00
75
76 void set_cache_policy(int cc)
77 {
78     unsigned int cp0;
79
80     cp0 = _mfc0(16, 0);
81     cp0 &= ~0x03;
82     cp0 |= cc;
83     _mtc0(16, 0, cp0);
84 }
85
86 int main(int argc, char** argv) {
87
88     set_cache_policy(WT_WA);
89
90     *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91     *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93     ks0 = *(volatile unsigned int*)0x80000300;
94     ks1 = *(volatile unsigned int*)0xA0000300;
95
96     while(1);
97
98     return (EXIT_SUCCESS);
99 }
100
```

Search Results		Output		Tasks		Usages	
	Name	Type	Address		Value		
	ks0	unsigned int	0x800002A8		0xDEADBEEF		
	ks1	unsigned int	0x800002A4		0xF005BA11		

A.4 Example 4: Write-through, No-write-allocate Writes to Data Memory

Write-through with no write-allocation means that the cache is only updated on a write if the corresponding line is already resident in the cache. If it is not resident, the data is written directly to physical memory, with no copy allocated to the cache. If we run our original program, the first write will not be allocated to cache, so when we do the KSEG0 read, we get the same data as the KSEG1 read (see [Figure A-12](#)).

FIGURE A-12:

```

68  volatile unsigned int ks0;
69  volatile unsigned int ks1;
70
71  #define UNCACHED 0x02
72  #define WB_WA    0x03
73  #define WT_WA    0x01
74  #define WT_NWA   0x00
75
76  void set_cache_policy(int cc)
77  {
78      unsigned int cp0;
79
80      cp0 = _mfc0(16, 0);
81      cp0 &= ~0x03;
82      cp0 |= cc;
83      _mtc0(16, 0, cp0);
84  }
85
86  int main(int argc, char** argv) {
87
88      set_cache_policy(WT_NWA);
89
90      *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
91      *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
92
93      ks0 = *(volatile unsigned int*)0x80000300;
94      ks1 = *(volatile unsigned int*)0xA0000300;
95
96      while(1);
97
98      return (EXIT_SUCCESS);
99  }

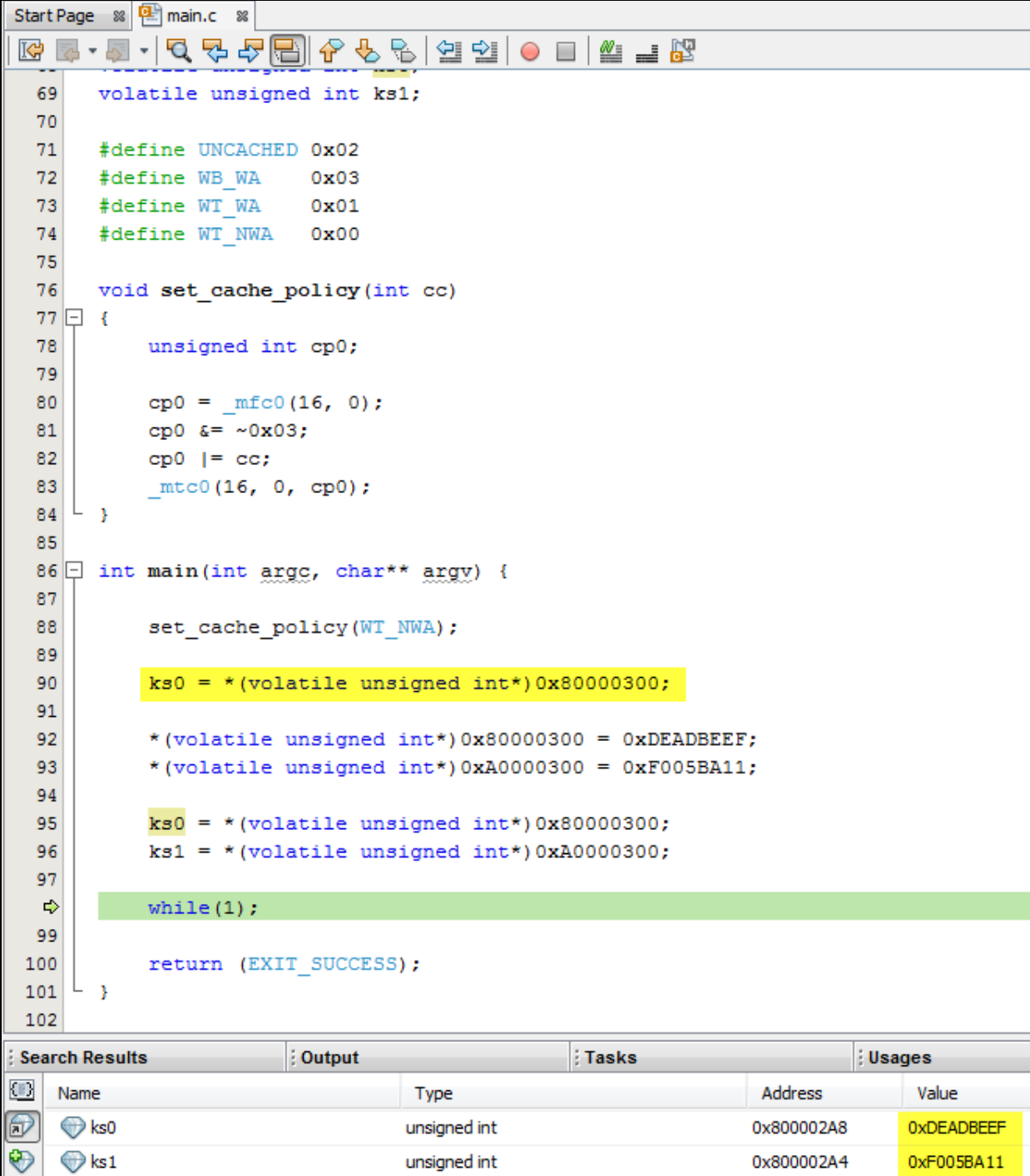
```

Search Results		Output		Tasks	Usages
Name	Type	Address	Value		
ks0	unsigned int	0x800002A8	0xF005BA11		
ks1	unsigned int	0x800002A4	0xF005BA11		

Using L1 Cache on PIC32MZ Devices

However, as shown in [Figure A-13](#), if we do a dummy read before the KSEG0 write, we get different results due to the read-allocation.

FIGURE A-13:



```
69 volatile unsigned int ks1;
70
71 #define UNCACHED 0x02
72 #define WB_WA 0x03
73 #define WT_WA 0x01
74 #define WT_NWA 0x00
75
76 void set_cache_policy(int cc)
77 {
78     unsigned int cp0;
79
80     cp0 = _mfc0(16, 0);
81     cp0 &= ~0x03;
82     cp0 |= cc;
83     _mtc0(16, 0, cp0);
84 }
85
86 int main(int argc, char** argv) {
87
88     set_cache_policy(WT_NWA);
89
90     ks0 = *(volatile unsigned int*)0x80000300;
91
92     *(volatile unsigned int*)0x80000300 = 0xDEADBEEF;
93     *(volatile unsigned int*)0xA0000300 = 0xF005BA11;
94
95     ks0 = *(volatile unsigned int*)0x80000300;
96     ks1 = *(volatile unsigned int*)0xA0000300;
97
98     while(1);
99
100     return (EXIT_SUCCESS);
101 }
102
```

Search Results		Output		Tasks		Usages	
	Name	Type	Address	Value			
	ks0	unsigned int	0x800002A8	0xDEADBEEF			
	ks1	unsigned int	0x800002A4	0xF005BA11			

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, flexPWR, JukeBlox, KEELOQ, KEELOQ logo, Klear, LANCheck, MediaLB, MOST, MOST logo, MPLAB, OptoLyzer, PIC, PICSTART, PIC³² logo, RightTouch, SpyNIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

The Embedded Control Solutions Company and mTouch are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, ECAN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, KlearNet, KlearNet logo, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICKit, PICtail, RightTouch logo, REAL ICE, SQI, Serial Quad I/O, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2013-2014, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-63276-364-8

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
= ISO/TS 16949 =

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
<http://www.microchip.com/support>
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110

Canada - Toronto
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880-3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Dusseldorf
Tel: 49-2129-3766400

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Pforzheim
Tel: 49-7231-424750

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Venice
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Poland - Warsaw
Tel: 48-22-3325737

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

03/25/14