UG647 June 24, 2015

Standalone (v.5.1)

Summary

Standalone is the lowest layer of software modules used to access processor specific functions. Standalone is used when an application accesses board/processor features directly and is below the operating system layer.

This document contains the following sections:

- MicroBlaze Processor API
- Cortex A9 Processor API
- Xilinx Hardware Abstraction Layer
- Program Profiling
- Configuring the Standalone OS
- MicroBlaze MMU Example

MicroBlaze Processor API

The following list is a summary of the MicroBlaze™ processor API sections. You can click on a link to go directly to the function section.

- MicroBlaze Processor Interrupt Handling
- MicroBlaze Processor Exception Handling
- MicroBlaze Processor Instruction Cache Handling
- MicroBlaze Processor Data Cache Handling
- MicroBlaze Processor Fast Simplex Link (FSL) Interface Macros
- MicroBlaze Processor FSL Macro Flags
- MicroBlaze Processor Pseudo-asm Macro Summary
- MicroBlaze Processor Version Register (PVR) Access Routine and Macros
- MicroBlaze Processor File Handling
- MicroBlaze Processor Errno

MicroBlaze Processor Interrupt Handling

The interrupt handling functions help manage interrupt handling on MicroBlaze processor devices. To use these functions, include the header file `mb_interface.h` in your source code.

MicroBlaze Processor Interrupt Handling Function Descriptions

---

`void` **`microblaze_enable_interrupts`**`(void)`

Enable interrupts on the MicroBlaze processor. When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on using this function.

`void` **`microblaze_disable_interrupts`**`(void)`

Disable interrupts on the MicroBlaze processor. This function can be called when entering a critical section of code where a context switch is undesirable.

`void` **`microblaze_register_handler`**`(XInterruptHandler` *`Handler`*`, void *`*`DataPtr`*`)`

Register the interrupt handler for the MicroBlaze processor. This handler is invoked in turn, by the first level interrupt handler that is present in Standalone.

The first level interrupt handler saves and restores registers, as necessary for interrupt handling, so that the function you register with this handler can be dedicated to the other aspects of interrupt handling, without the overhead of saving and restoring registers.

MicroBlaze Processor Exception Handling

This section describes the exception handling functionality available on the MicroBlaze processor. This feature and the corresponding interfaces are not available on versions of the MicroBlaze processor older than v3.00.a.

Note: These functions work correctly only when the parameters that determine hardware exception handling are configured appropriately in the MicroBlaze Microprocessor Hardware Specification (MHS) hardware block. For example, you can register a handler for divide by zero exceptions only if hardware divide by zero exceptions are enabled on the MicroBlaze processor. Refer to the *MicroBlaze Processor Reference Guide (UG081)* for information on how to configure these cache parameters. A link to that document can be found in "MicroBlaze Processor API," page 1.

MicroBlaze Processor Exception Handler Function Descriptions

The following functions help manage exceptions on the MicroBlaze processor. You must include the `mb_interface.h` header file in your source code to use these functions.

`void` **`microblaze_disable_exceptions`**`(void)`

Disable hardware exceptions from the MicroBlaze processor. This routine clears the appropriate "exceptions enable" bit in the model-specific register (MSR) of the processor.

`void` **`microblaze_enable_exceptions`**`(void)`

Enable hardware exceptions from the MicroBlaze processor. This routine sets the appropriate "exceptions enable" bit in the MSR of the processor.

`void` **`microblaze_register_exception_handler`**`(u8` *`ExceptionId`*`, XExceptionHandler` *`Handler`*`, void *`*`DataPtr`*`)`

Register a handler for the specified exception type. *`Handler`* is the function that handles the specified exception.
`DataPtr` is a callback data value that is passed to the exception handler at run-time. By default the exception ID of the corresponding exception is passed to the handler.

Table 1 describes the valid exception IDs, which are defined in the `microblaze_exceptions_i.h` file.

Table 1: **Valid Exception IDs**

Exception ID	Value	Description
`XEXC_ID_FSL`	0	FSL bus exceptions.
`XEXC_ID_UNALIGNED_ACCESS`	1	Unaligned access exceptions.
`XEXC_ID_ILLEGAL_OPCODE`	2	Exception due to an attempt to execute an illegal opcode.
`XEXC_ID_M_AXI_I_EXCEPTION(1)`	3	Exception due to a timeout from the Instruction side system bus.
`XEXC_ID_M_AXI_D_EXCEPTION(1)`	4	Exception due to a timeout on the Data side system bus.
`XEXC_ID_DIV_BY_ZERO`	5	Divide by zero exceptions from the hardware divide.
`XEXC_ID_FPU`	6	Exceptions from the floating point unit on the MicroBlaze processor. ***Note:*** This exception is valid only on v4.0 and later versions of the MicroBlaze processor.
`XEXC_ID_MMU`	7	Exceptions from the MicroBlaze processor MMU. All possible MMU exceptions are vectored to the same handler. ***Note:*** This exception is valid only on v7.00.a and later versions of the MicroBlaze processor.

By default, Standalone provides empty, no-op handlers for all the exceptions *except* unaligned exceptions. A default, fast, unaligned access exception handler is provided by Standalone.

An unaligned exception can be handled by making the corresponding aligned access to the appropriate bytes in memory. Unaligned access is transparently handled by the default handler. However, software that makes a significant amount of unaligned accesses will see the performance effects of this at run-time. This is because the software exception handler takes much longer to satisfy the unaligned access request as compared to an aligned access.

In some cases you might want to use the provision for unaligned exceptions to just trap the exception, and to be aware of what software is causing the exception. In this case, you should set breakpoints at the unaligned exception handler, to trap the dynamic occurrence of such an exception or register your own custom handler for unaligned exceptions.

Note: The lowest layer of exception handling, always provided by Standalone, stores volatile and temporary registers on the stack; consequently, your custom handlers for exceptions must take into consideration that the first level exception handler will have saved some state on the stack, before invoking your handler.

Nested exceptions are allowed by the MicroBlaze processor. The exception handler, in its prologue, re-enables exceptions. Thus, exceptions within exception handlers are allowed and handled. When the `predecode_fpu_exceptions` parameter is set to `true`, it causes the low-level exception handler to:

- Decode the faulting floating point instruction
- Determine the operand registers
- Store their values into two global variables

You can register a handler for floating point exceptions and retrieve the values of the operands from the global variables. You can use the `microblaze_getfpex_operand_a()` and `microblaze_getfpex_operand_b()` macros.

Note: These macros return the operand values of the last floating point (FP) exception. If there are nested exceptions, you cannot retrieve the values of outer exceptions. An FP instruction might have one of the source registers being the same as the destination operand. In this case, the faulting instruction overwrites the input operand value and it is again irrecoverable.

MicroBlaze Processor Instruction Cache Handling

The following functions help manage instruction caches on the MicroBlaze processor. You must include the `xil_cache.h` header file in your source code to use these functions.

Note: These functions work correctly only when the parameters that determine the caching system are configured appropriately in the MicroBlaze Microprocessor Hardware Specification (MHS) hardware block. Refer to the *MicroBlaze Reference Guide (UG081)* for information on how to configure these cache parameters. "MicroBlaze Processor API," page 1 contains a link to this document.

MicroBlaze Processor Instruction Cache Handling Function Descriptions

---

`void` **`Xil_ICacheEnable`**`(void)`

Enable the instruction cache on the MicroBlaze processor. When the MicroBlaze processor starts up, the instruction cache is disabled. The instruction cache must be explicitly turned on using this function.

`void` **`Xil_ICacheDisable`**`(void)`

Disable the instruction cache on the MicroBlaze processor.

`void` **`Xil_ICacheInvalidate`**`()`

Invalidate the instruction icache.

Note: For MicroBlaze processors prior to version v7.20.a, the cache and interrupts are disabled before invalidation starts and restored to their previous state after invalidation.

`void` **`Xil_ICacheInvalidateRange`**`(unsigned int` *`cache_addr`*`,`
`unsigned int` *`cache_size`*`)`

MicroBlaze Processor Data Cache Handling

The following functions help manage data caches on the MicroBlaze processor. You must include the header file `xil_cache.h` in your source code to use these functions.

Note: These functions work correctly only when the parameters that determine the caching system are configured appropriately in the MicroBlaze MHS hardware block. Refer to the *MicroBlaze Processor Reference Guide (UG081)* for information on how to configure these cache parameters. "MicroBlaze Processor API," page 1 contains a link to this document.

Data Cache Handling Functions

`void` **`Xil_DCacheEnable`**`(void)`

Enable the data cache on the MicroBlaze processor. When the MicroBlaze processor starts up, the data cache is disabled. The data cache must be explicitly turned on using this function.

`void` **`Xil_DCache_Disable`**`(void)`

Disable the data cache on the MicroBlaze processor. If writeback caches are enabled in the MicroBlaze processor hardware, this function also flushes the dirty data in the cache back to external memory and invalidates the cache. For write through caches, this function does not do any extra processing other than disabling the cache.

If the L2 cache system is present in the hardware, this function flushes the L2 cache before disabling the DCache.

`void` **`Xil_DCacheFlush`**`()`

Flush the entire data cache. This function can be used when write-back caches are turned on in the MicroBlaze processor hardware. Executing this function ensures that the dirty data in the cache is written back to external memory and the contents invalidated.

If the L2 cache system is present in the hardware, this function flushes the L2 cache first, before flushing the L1 cache.

`void` **`Xil_DCacheFlushRange`**`(unsigned int` *`cache_addr`*`,`
`unsigned int` *`cache_len`*`)`

Flush the specified data cache range. This function can be used when write-back caches are enabled in the MicroBlaze processor hardware. Executing this function ensures that the dirty data in the cache range is written back to external memory and the contents of the cache range are invalidated. Note that *cache lines* will be flushed starting from the cache line to which cache_addr belongs and ending at the cache line containing the address (*`cache_addr`* + *`cache_size`* - 1).

If the L2 cache system is present in the hardware, this function flushes the relevant L2 cache range first, before flushing the L1 cache range.

For example, `Xil_DCacheFlushRange (0x00000300, 0x100)` flushes the data cache region from 0x300 to 0x3ff (0x100 bytes of cache memory is flushed starting from 0x300).

`void` **`Xil_DCacheInvalidate`**`()`

Invalidate the data cache.

If the L2 cache system is present in the hardware, this function invalidates the L2 cache first, before invalidating the L1 cache.

Note: For MicroBlaze processors prior to version v7.20.a, the cache and interrupts are disabled before invalidation starts and restored to their previous state after invalidation.

`void` **`Xil_DCacheInvalidateRange`**`(unsigned int` *`cache_addr`*`, unsigned int` *`cache_size`*

Invalidate the data cache. This function can be used for invalidating all or part of the data cache. The parameter *`cache_addr`* indicates the beginning of the cache location and *`cache_size`* represents the size from *`cache_addr`* to invalidate.

Note that *cache lines* will be invalidated starting from the cache line to which *cache_addr* belongs and ending at the cache line containing the address (*`cache_addr`* + *`cache_size`* - 1).

If the L2 cache system is present in the hardware, this function invalidates the relevant L2 cache range first, before invalidating the L1 cache range.

Note: For MicroBlaze processors prior to version v7.20.a, the cache and interrupts are disabled before invalidation starts and restored to their previous state after invalidation.

For example, `Xil_DCacheInvalidateRange` (0x00000300, 0x100) invalidates the data cache region from 0x300 to 0x3ff (0x100 bytes of cache memory is cleared starting from 0x300).

Software Sequence for Initializing Instruction and Data Caches

Typically, before using the cache, your program must perform a particular sequence of cache operations to ensure that invalid/dirty data in the cache is not being used by the processor. This would typically happen during repeated program downloads and executions.

The following example snippets show the necessary software sequence for initializing instruction and data caches in your program.

```
/* Initialize ICache *//
Xil_ICacheInvalidate ();
Xil_ICacheEnable ();

/* Initialize DCache */
Xil_DCacheInvalidate ();
Xil_DCacheEnable ();
```

At the end of your program, you should also put in a.3(5.6r)-1.49.1(f8b0038 ,i4.5(itia)4.w()-6(a)5.13.9(
Xil_ICacheIn ();

MicroBlaze Processor Fast Simplex Link (FSL) Interface Macros

Standalone includes macros to provide convenient access to accelerators connected to the MicroBlaze Fast Simplex Link (FSL) Interfaces.

MicroBlaze Processor Fast Simplex Link (FSL) Interface Macro Summary

The following is a list of the available macros. Click on a macro name to go to the description of the active macros.

MicroBlaze Processor FSL Macro Descriptions

The following macros provide access to all of the functionality of the MicroBlaze FSL feature in one simple and parameterized interface. Some capabilities are available on MicroBlaze v7.00.a and later only, as noted in the descriptions.

In the macro descriptions, *val* refers to a variable in your program that can be the source or sink of the FSL operation.

Note: *id* must be an integer *literal* in the basic versions of the macro (`getfslx, putfslx, tgetfslx, tputfslx`) and can be an integer literal or an integer variable in the dynamic versions of the macros (`getdfslx, putdfslx, tgetdfslx, tputdfslx.`)

You must include `fsl.h` in your source files to make these macros available.

---

getfslx(*val,id,flags*)

Performs a get function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is a literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later). The semantics of the instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9.

---

putfslx(*val,id,flags*)

Performs a put function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is a literal in the range of 0 to 7 (0 to 15 for MicroBlaze processor v7.00.a and later).

The semantics of the instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9.

---

tgetfslx(*val,id,flags*)

Performs a test get function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is a literal in the ranging of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later). This macro can be used to test reading a single value from the FSL. The semantics of the instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9.

---

tputfslx(*val,id,flags*)

Performs a put function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is a literal in the range of 0 to 7 (0 to 15 for MicroBlaze processor v7.00.a and later). This macro can be used to test writing a single value to the FSL.The semantics of the put instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9.

getd fslx(*val,id,flags*)

Performs a get function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is an integer value or variable in the range of 0 to 15. The semantics of the instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9. This macro is available on MicroBlaze processor v7.00.a and later only.

---

putdfslx(*val,id,flags*)

Performs a put function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is an integer value or variable in the range of 0 to 15. The semantics of the instruction is determined by the valid FSL macro flags, which are listed in Table 2, page 9. This macro is available on MicroBlaze processor v7.00.a and later only.

---

tgetdfslx(*val,id,flags*)

Performs a test get function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is an integer or variable in the range of 0 to 15. This macro can be used to test reading a single value from the FSL. The semantics of the instruction is determined by the valid FSL macro flags, listed in Table 2. This macro is available on MicroBlaze processor v7.00.a and later only.

---

tputdfslx(*val,id,flags*)

Performs a put function on an input FSL of the MicroBlaze processor; id is the FSL identifier and is an integer or variable in the range of 0 to 15. This macro can be used to test writing a single value to the FSL.The semantics of the instruction is determined by the valid FSL macro flags, listed in Table 2. This macro is available on MicroBlaze processor v7.00.a and later only.

---

fsl_isinvalid(*invalid*)

Checks if the last FSL operation returned valid data. This macro is applicable after invoking a non-blocking FSL put or get instruction. If there was no data on the FSL channel on a get, or if the FSL channel was full on a put, *invalid* is set to 1; otherwise, it is set to 0.

---

fsl_iserror(*error*)

This macro is used to check if the last FSL operation set an error flag. This macro is applicable after invoking a control FSL put or get instruction. If the control bit was set *error* is set to 1; otherwise, it is set to 0.

MicroBlaze Processor FSL Macro Flags

Table 2 lists the available FSL Macro flags.

Deprecated MicroBlaze Processor Fast Simplex Link (FSL) Macros

The following macros are deprecated:

---

getfsl(*val,id*)(deprecated)

Performs a blocking data get function on an input FSL of the MicroBlaze processor;

ngetfsl(*val,id*)(deprecated)

Performs a non-blocking data get function on an input FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7.

---

nputfsl(*val,id*)(deprecated)

Performs a non-blocking data put function on an output FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7.

---

cgetfsl(*val, id*)(deprecated)

Performs a blocking control get function on an input FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7. This macro is uninterruptible.

---

cputfsl(*val, id*)(deprecated)

Performs a blocking control put function on an output FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7. This macro is uninterruptible.

---

ncgetfsl(*val, id*)(deprecated)

Performs a non-blocking control get function on an input FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7.

---

ncputfsl(*val, id*)(deprecated)

Performs a non-blocking control put function on an output FSL of the MicroBlaze processor; *id* is the FSL identifier in the range of 0 to 7.

---

getfsl_interruptible(*val, id*)(deprecated)

Performs repeated non-blocking data get operations on an input FSL of the MicroBlaze processor until valid data is actually fetched; *id* is the FSL identifier in the range of 0 to 7. Because the FSL access is non-blocking, interrupts will be serviced by the processor.

---

putfsl_interruptible(*val, id*)(deprecated)

Performs repeated non-blocking data put operations on an output FSL of the MicroBlaze processor until valid data is sent out; *id* is the FSL identifier in the range of 0 to 7. Because the FSL access is non-blocking, interrupts will be serviced by the processor.

cgetfsl_interruptible(*val, id*)(deprecated)

Performs repeated non-blocking control get operations on an input FSL of the MicroBlaze processor until valid data is actually fetched; *id* is the FSL identifier in the range of 0 to 7. Because the FSL access is non-blocking, interrupts are serviced by the processor.

---

cputfsl_interruptible(*val, id*)(deprecated)

Performs repeated non-blocking control put operations on an output FSL of the MicroBlaze processor until valid data is sent out; *id* is the FSL identifier in the range of 0 to 7. Because the FSL access is non-blocking, interrupts are serviced by the processor.

MicroBlaze Processor Pseudo-asm Macros

Standalone includes macros to provide convenient access to various registers in the MicroBlaze processor. Some of these macros are very useful within exception handlers for retrieving information about the exception. To use these macros, you must include the `mb_interface.h` header file in your source code.

MicroBlaze Processor Pseudo-asm Macro Summary

The following is a summary of the MicroBlaze processor pseudo-asm macros. Click on the macro name to go to the description.

MicroBlaze Processor Pseudo-asm Macro Descriptions

---

mfgpr(*rn*)

Return value from the general purpose register (GPR) *rn*.

---

mfmsr( )

Return the current value of the MSR.

---

mfesr( )

Return the current value of the Exception Status Register (ESR).

---

mfear( )

Return the current value of the Exception Address Register (EAR).

mffsr( )

Return the current value of the Floating Point Status (FPS).

---

mtmsr(*v*)

Move the value v to MSR.

---

mtgpr(*rn, v*)

Move the value *v* to GPR *rn*.

---

microblaze_getfpex_operand_a( )

Return the saved value of operand A of the last faulting floating point instruction.

---

microblaze_getfpex_operand_b( )

Return the saved value of operand B of the last faulting floating point instruction.

Note: Because of the way some of these macros have been written, they cannot be used as parameters to function calls and other such constructs.

---

clz(v)

Counts the number of leading zeros in the data specified by v

---

mbar(mask)

This instruction ensures that outstanding memory accesses on memory interfaces are completed before any subsequent instructions are executed. mask value of 1 specifies data side barrier, mask value of 2 specifies instruction side barrier and mask value of 16 specifies to put the processor in sleep.

---

mb_swapb(v)

Swaps the bytes in the data specified bv v. This converts the bytes in the data from little endian to big endian or vice versa. So v contains a value of 0x12345678, the macro will return a value of 0x78563412.

---

mb_swaph(v)

Swaps the half words in the data specified bv v. So if v has a value of 0x12345678, the macro will return a value of 0x56781234.

---

mb_sleep

Puts the processor in sleep.

MicroBlaze Processor Version Register (PVR) Access Routine and Macros

MicroBlaze processor v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the *pvr_t* data structure, which is defined as an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the *pvr_t* data structure is resized to hold only as many PVRs as are present in hardware.

To access information in the PVR:

1. Use the `microblaze_get_pvr()` function to populate the PVR data into a *pvr_t* data structure.
2. In subsequent steps, you can use any one of the PVR access macros list to get individual data stored in the PVR.

Note: The PVR access macros take a parameter, which must be of type *pvr_t*.

PVR Access Routine

The following routine is used to access the PVR. You must include `pvr.h` file to make this routine available.

`int` **`microblaze_get_pvr`**`(`*`pvr_t *pvr`*`)`

Populate the PVR data structure to which *pvr* points with the values of the hardware PVR registers. This routine populates only as many PVRs as are present in hardware and the rest are zeroed. This routine is not available if `C_PVR` is set to NONE in hardware.

PVR Macros

The following processor macros are used to access the PVR. You must include `pvr.h` file to make these macros available.

Table 3 lists the MicroBlaze processor PVR macros and descriptions.

Table 3: **PVR Access Macros**

Macro	Description
`MICROBLAZE_PVR_IS_FULL(pvr)`	Return non-zero integer if PVR is of type FULL, 0 if basic.
`MICROBLAZE_PVR_USE_BARREL(pvr)`	Return non-zero integer if hardware barrel shifter present.
`MICROBLAZE_PVR_USE_DIV(pvr)`	Return non-zero integer if hardware divider present.
`MICROBLAZE_PVR_USE_HW_MUL(pvr)`	Return non-zero integer if hardware multiplier present.
`MICROBLAZE_PVR_USE_FPU(pvr)`	Return non-zero integer if hardware floating point unit (FPU) present.
`MICROBLAZE_PVR_USE_FPU2(pvr)`	Return non-zero integer if hardware floating point conversion and square root instructions are present.
`MICROBLAZE_PVR_USE_ICACHE(pvr)`	Return non-zero integer if I-cache present.
`MICROBLAZE_PVR_USE_DCACHE(pvr)`	Return non-zero integer if D-cache present.

Table 3: **PVR Access Macros *(Cont'd)***

Macro	Description
`MICROBLAZE_PVR_MICROBLAZE_VERSION (pvr)`	Return MicroBlaze processor version encoding. Refer to the *MicroBlaze Processor Reference Guide (UG081)* for mappings from encodings to actual hardware versions. "MicroBlaze Processor API," page 1 contains a link to this document.
`MICROBLAZE_PVR_USER1(pvr)`	Return the `USER1` field stored in the PVR.
`MICROBLAZE_PVR_USER2(pvr)`	Return the `USER2` field stored in the PVR.
`MICROBLAZE_PVR_INTERCONNECT(pvr)`	Return non-zero if MicroBlaze processor has PLB interconnect; otherwise return zero.
`MICROBLAZE_PVR_D_PLB(pvr)`	Return non-zero integer if Data Side PLB interface is present.
`MICROBLAZE_PVR_D_OPB(pvr)`	Return non-zero integer if Data Side On-chip Peripheral Bus (OPB) interface present.
`MICROBLAZE_PVR_D_LMB(pvr)`	Return non-zero integer if Data Side Local Memory Bus (LMB) interface present.
`MICROBLAZE_PVR_I_PLB(pvr)`	Return non-zero integer if Instruction Side PLB interface is present.
`MICROBLAZE_PVR_I_OPB(pvr)`	Return non-zero integer if Instruction side OPB interface present.
`MICROBLAZE_PVR_I_LMB(pvr)`	Return non-zero integer if Instruction side LMB interface present.
`MICROBLAZE_PVR_INTERRUPT_IS_EDGE (pvr)`	Return non-zero integer if interrupts are configured as edge-triggered.
`MICROBLAZE_PVR_EDGE_IS_POSITIVE (pvr)`	Return non-zero integer if interrupts are configured as positive edge triggered.
`MICROBLAZE_PVR_USE_MUL64(pvr)`	Return non-zero integer if MicroBlaze processor supports 64-bit products for multiplies.
`MICROBLAZE_PVR_OPCODE_OxO_ILLEGAL (pvr)`	Return non-zero integer if opcode 0x0 is treated as an illegal opcode.
`MICROBLAZE_PVR_UNALIGNED_EXCEPTION (pvr)`	Return non-zero integer if unaligned exceptions are supported.
`MICROBLAZE_PVR_ILL_OPCODE_ EXCEPTION(pvr)`	Return non-zero integer if illegal opcode exceptions are supported.
`MICROBLAZE_PVR_IOPB_EXCEPTION(pvr)`	Return non-zero integer if I-OPB exceptions are supported.
`MICROBLAZE_PVR_DOPB_EXCEPTION(pvr)`	Return non-zero integer if D-OPB exceptions are supported.
`MICROBLAZE_PVR_IPLB_EXCEPTION(pvr)`	Return non-zero integer if I-PLB exceptions are supported.
`MICROBLAZE_PVR_DPLB_EXCEPTION(pvr)`	Return non-zero integer if D-PLB exceptions are supported.
`MICROBLAZE_PVR_DIV_ZERO_EXCEPTION (pvr)`	Return non-zero integer if divide by zero exceptions are supported.

Table 3: **PVR Access Macros *(Cont'd)***

Macro	Description
MICROBLAZE_PVR_FPU_EXCEPTION(pvr)	Return non-zero integer if FPU exceptions are supported.
MICROBLAZE_PVR_FSL_EXCEPTION(pvr)	Return non-zero integer if FSL exceptions are present.
MICROBLAZE_PVR_DEBUG_ENABLED(pvr)	Return non-zero integer if debug is enabled.
MICROBLAZE_PVR_NUM_PC_BRK(pvr)	Return the number of hardware PC breakpoints available.
MICROBLAZE_PVR_NUM_RD_ADDR_BRK (pvr)	Return the number of read address hardware watchpoints supported.
MICROBLAZE_PVR_NUM_WR_ADDR_BRK (pvr)	Return the number of write address hardware watchpoints supported.
MICROBLAZE_PVR_FSL_LINKS(pvr)	Return the number of FSL links present.
MICROBLAZE_PVR_ICACHE_BASEADDR (pvr)	Return the base address of the I-cache.
MICROBLAZE_PVR_ICACHE_HIGHADDR (pvr)	Return the high address of the I-cache.
MICROBLAZE_PVR_ICACHE_ADDR_TAG_ BITS(pvr)	Return the number of address tag bits for the I-cache.
MICROBLAZE_PVR_ICACHE_USE_FSL(pvr)	Return non-zero if I-cache uses FSL links.
MICROBLAZE_PVR_ICACHE_ALLOW_WR (pvr)	Return non-zero if writes to I-caches are allowed.
MICROBLAZE_PVR_ICACHE_LINE_LEN (pvr)	Return the length of each I-cache line in bytes.
MICROBLAZE_PVR_ICACHE_BYTE_SIZE (pvr)	Return the size of the D-cache in bytes.
MICROBLAZE_PVR_DCACHE_BASEADDR (pvr)	Return the base address of the D-cache.
MICROBLAZE_PVR_DCACHE_HIGHADDR (pvr)	Return the high address of the D-cache.
MICROBLAZE_PVR_DCACHE_ADDR_TAG_ BITS(pvr)	Return the number of address tag bits for the D-cache.
MICROBLAZE_PVR_DCACHE_USE_FSL(pvr)	Return non-zero if the D-cache uses FSL links.
MICROBLAZE_PVR_DCACHE_ALLOW_WR (pvr)	Return non-zero if writes to D-cache are allowed.
MICROBLAZE_PVR_DCACHE_LINE_LEN (pvr)	Return the length of each line in the D-cache in bytes.
MICROBLAZE_PVR_DCACHE_BYTE_SIZE (pvr)	Return the size of the D-cache in bytes.
MICROBLAZE_PVR_TARGET_FAMILY(pvr)	Return the encoded target family identifier.

Table 3: **PVR Access Macros *(Cont'd)***

Macro	Description
MICROBLAZE_PVR_MSR_RESET_VALUE	Refer to the *MicroBlaze Processor Reference Guide (UG081)* for mappings from encodings to target family name strings. "MicroBlaze Processor API," page 1 contains a link to this document.
MICROBLAZE_PVR_MMU_TYPE(pvr)	Returns the value of C_USE_MMU. Refer to the *MicroBlaze Processor Reference Guide (UG081)* for mappings from MMU type values to MMU function. "MicroBlaze Processor API," page 1 contains a link to this document.

MicroBlaze Processor File Handling

The following routine is included for file handling:

`int` **`fcntl`**`(int` *`fd, int cmd, long arg`*`);`

A dummy implementation of `fcntl( )`, which always returns 0, is provided. **`fcntl`** is intended to manipulate file descriptors according to the command specified by *`cmd.`* Because Standalone does not provide a file system, this function is included for completeness only.

MicroBlaze Processor Errno

The following routine provides the error number value:

`int` **`errno`**`( );`

Return the global value of errno as set by the last C library call.

Cortex A9 Processor API

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compilers.

The following lists the Cortex A9 Processor API sections.You can click on a link to go directly to the function section.

The following subsections describe the functions by type.

Cortex A9 Processor Boot Code

The boot.S file contains a minimal set of code for transferring control from the processor's reset location to the start of the application. It performs the following tasks.

- Invalidate L1 caches, TLBs, Branch Predictor Array, etc.
- Invalidate L2 caches and initialize L2 Cache Controller
- Enable caches and MMU
- Load MMU translation table base address into the TTB registers
- Enable NEON coprocessor

The boot code also starts the Cycle Counter and initializes the Static Memory Controller.

Cortex A9 Processor Cache Functions

The xil_cache.c file and the corresponding xil_cache.h header file provide access to the following cache and cache-related operations.

Cache Function Summary

The following are links to the function descriptions. Click on the name to go to that function.

Cache Function Descriptions

`void` **`Xil_DCacheEnable`**`(void)`

Enable the data caches.

`void` **`Xil_DCacheInvalidate`**`(void)`

Invalidate the entire data cache.

`void` **`Xil_DCacheInvalidateLine`**`(unsigned int` *`adr`*`)`

Invalidate a data cache line. If the byte specified by adr is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated. A subsequent data access to this address results in a cache miss and a cache line refill.

`void` **`Xil_ICacheInvalidateRange`**`(unsigned int` *`adr`*`, unsigned` *`len`*`)`

Invalidate the instruction cache for the given address range. If the bytes specified by the adr are cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L1DCacheEnable`**`(void)`

Enable the level 1 data cache.

`void` **`Xil_L1DCacheDisable`**`(void)`

Disable the level 1 data cache.

`void` **`Xil_L1DCacheInvalidate`**`(void)`

Invalidate the level 1 data cache.

`void` **`Xil_L1DCacheInvalidateLine`**`(unsigned int` *`adr`*`)`

Invalidate a level 1 data cache line. If the byte specified by the adr is cached by the data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L1DCacheInvalidateRange`**`(unsigned int` *`adr`*`, unsigned` *`len`*`)`

Invalidate the level 1 data cache for the given address range. If the bytes specified by the adr are cached by the sata cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L1DCacheFlush`**`(void)`

Flush the level 1 data cache.

`void` **`Xil_L1DCacheFlushLine`**`(unsigned int` *`adr`*`)`

Flush a level 1 data cache line. If the byte specified by the adr is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

`void` **`Xil_L1DCacheFlushRange`**`(unsigned int` *`adr`*`, unsigned` *`len`*`)`

Flush the level 1 data cache for the given address range. If the bytes specified by the adr are cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the written to system memory first before the before the line is invalidated.

`void` **`Xil_L1DCacheStoreLine`**`(unsigned int` *`adr`*`)`

Store a level 1 data cache line. If the byte specified by the adr is cached by the data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

`void` **`Xil_L1ICacheEnable`**`(void)`

Enable the level 1 instruction cache.

`void` **`Xil_L1ICacheDisable`**`(void)`

Disable level 1 the instruction cache.

`void` **`Xil_L1ICacheInvalidate`**`(void)`

Invalidate the entire level 1 instruction cache.

`void` **`Xil_L1ICacheInvalidateLine`**`(unsigned int` *`adr`*`)`

Invalidate a level 1 instruction cache line. If the instruction specified by the parameter adr is cached by the instruction cache, the cacheline containing that instruction is invalidated.

`void` **`Xil_L1ICacheInvalidateRange`**`(unsigned int` *`adr`*`,`
`unsigned` *`len`*`)`

Invalidate the level 1 instruction cache for the given address range. If the bytes specified by the adr are cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L2CacheEnable`**`(void)`

Enable the L2 cache.

`void` **`Xil_L2CacheDisable`**`(void)`

Disable the L2 cache.

`void` **`Xil_L2CacheInvalidate`**`(void)`

Invalidate the L2 cache. If the byte specified by the adr is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L2CacheInvalidateLine`**`(unsigned int` *`adr`*`)`

Invalidate a level 2 cache line. If the byte specified by the adr is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L2CacheInvalidateRange`**`(unsigned int` *`adr`*`, unsigned` *`len`*`)`

Invalidate the level 2 cache for the given address range. If the bytes specified by the adr are cached by the data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost and are not written to system memory before the line is invalidated.

`void` **`Xil_L2CacheFlush`**`(void)`

Flush the L2 cache. If the byte specified by the adr is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

`void` **`Xil_L2CacheFlushLine`**`(unsigned int` *`adr`*`)`

Flush a level 1 cache line. If the byte specified by the adr is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

`void` **`Xil_L2CacheFlushRange`**`adrlen)unsigned`

Cortex A9 Processor MMU Handling

The standalone BSP MMU handling API is implemented in file `xil_mmu.c` and the corresponding header `file xil_mmu.h`.

MMU Handling Function Summary

The following function describes the available MMU handling API.

`void Xil_SetTlbAttributes``(u32 addr, u32 attrib)`

This function changes the MMU attribute of the 1 MB address range in which the passed memory address "addr" falls.

The new MMU attribute is passed as an argument "attrib" to this API.

This API can be used to change attributes such as cache-ability and share-ability of a specified memory region.

Cortex A9 Processor Exception Handling

The Standalone BSP provides an exception handling API. For details about the exceptions and interrupts on ARM Cortex-A9 processor, refer to "Exceptions" under the chapter "The System Level Programmers' Model" in the ARM Architecture Reference Manual ARMv7-A and ARMv-7R edition.

The exception handling API is implemented in a set of the files - asm_vectors.S, vectors.c, xil_exception.c, and the corresponding header files vectors.h and xil_exception.h.

Exception Handling Function Summary

The following are links to the function descriptions. Click on the name to go to that function.

void Xil_ExceptionInit(void)

void Xil_ExceptionRegisterHandler (u8 ExceptionId, XExceptionHandler Handler, void *DataPtr)

void Xil_ExceptionRemoveHandler (u8 ExceptionId)

void Xil_ExceptionEnableMask(Mask)

void Xil_ExceptionEnable(void)

void Xil_ExceptionDisableMask(Mask)

void Xil_ExceptionDisable(void)

Exception Handling Function Descriptions

`void` **`Xil_ExceptionInit`**`(void)`

Sets up the interrupt vector table and registers a "do nothing" function for each exception. This function has no parameters and does not return a value. This function must be called before registering any exception handlers or enabling any interrupts.

void **Xil_ExceptionRegisterHandler** (u8 ExceptionId, XExceptionHandler Handler, void *DataPtr)

Registers an exception handler for a specific exception; does not return a value. Refer to Table 1, for a list of exception types and their values.

The parameters are:

- **ExceptionId** is of parameter type u8, and is the exception to which this handler should be registered. The type and the values are defined in the xil_exception.h header file.
- **Handler** is an Xil_ExceptionHandler parameter that is the pointer to the exception handling function.

  The function provided as the Handler parameter must have the following function prototype:

  typedef void (*Xil_ExceptionHandler)(void * DataPtr);

  This prototype is declared in the xil_exception.h header file.
- **DataPtr** is of parameter type void * and is the user value to be passed when the Handler is called.

  When this Handler function is called, the parameter DataPtr contains the same value provided, when the Handler was registered.

Table 4: Registered Exception Types and Values

Exception Type	Value
XIL_EXCEPTION_ID_RESET	0
XIL_EXCEPTION_ID_UNDEFINED_INT	1
XIL_EXCEPTION_ID_SWI_INT	2
XIL_EXCEPTION_ID_PREFETCH_ABORT_INT	3
XIL_EXCEPTION_ID_DATA_ABORT_INT	4
XIL_EXCEPTION_ID_IRQ_INT	5
XIL_EXCEPTION_ID_FIQ_INT	6

---

void **Xil_ExceptionRemoveHandler** (u8 ExceptionId)

De-register a handler function for a given exception. For possible values of parameter ExceptionId, refer to Table 1.

---

void **Xil_ExceptionEnableMask**(Mask)

Enable exceptions specified by Mask. The parameter Mask is a bitmask for exceptions to be enabled. The Mask parameter can have the values XIL_EXCEPTION_IRQ, XIL_EXCEPTION_FIQ, or XIL_EXCEPTION_ALL.

---

void **Xil_ExceptionEnable**(void)

Enable the IRQ exception.

These macros must be called after initializing the vector table with function Xil_exceptionInit and registering exception handlers with function Xil_ExceptionRegisterHandler.

`void` **`Xil_ExceptionDisableMask`**`(Mask)`

Disable exceptions specified by Mask. The parameter Mask is a bitmask for exceptions to be disabled. The Mask parameter can have the values XIL_EXCEPTION_IRQ, XIL_EXCEPTION_FIQ, or XIL_EXCEPTION_ALL.

`void` **`Xil_ExceptionDisable`**`(void)`

Disable the IRQ exception.

Cortex A9 Processor and pl310 Errata Support

Various ARM errata are handled in the standalone BSP. The implementation for errata handling follows ARM guidelines and is based on the open source Linux support for these errata. The errata conditions handled in the standalone BSP are listed below.

- ARM erratum number 742230 (DMB operation may be faulty)
- ARM erratum number 743622 (Faulty hazard checking in the Store Buffer may lead to data corruption)
- ARM erratum number 775420 (A data cache maintenance operation which aborts, might lead to deadlock)
- ARM erratum number 794073 (Speculative instruction fetches with MMU disabled might not comply with architectural requirements)
- ARM erratum number 588369 (Clean & Invalidate maintenance operations do not invalidate clean lines)
- ARM PL310 erratum number 727915 (Background Clean and Invalidate by Way operation can cause data corruption)
- ARM PL310 erratum number 753970 (Cache sync operation may be faulty)

For further information on these errata items, please refer to the appropriate ARM documentation at ARM the information center.

The BSP file `xil_errata.h` defines macros for these errata. The handling of the errata are enabled by default. To disable handling of all the errata globally, un-define the macro `ENABLE_ARM_ERRATA` in `xil_errata.h`. To disable errata on a per-erratum basis, un-define relevant macros in `xil_errata.h`.

Cortex A9 Processor File Support

The following links take you directly to the `gcc` file support function.

int read(int fd, char *buf, int nbytes)
int write(int fd, char *buf, int nbytes)
int isatty(int fd)
int fcntl (int fd, int cmd, long arg)

File support is limited to the `stdin` and `stdout` streams. Consequently, the following functions are *not* necessary:

gcc

- `open( ) (in gcc/open.c)`
- `close( ) (in gcc/close.c)`
- `fstat( ) (in gcc/fstat.c)`
- `unlink( ) (in gcc/unlink.c)`
- `lseek( ) (in gcc/lseek.c)`

These files are included for completeness and because they are referenced by the C library.

Cortex A9 gcc File Support Function Descriptions

`int` **`read`**`(int fd, char *buf, int nbytes)`

The read( ) function in gcc/read.c reads nbytes bytes from the standard input by calling inbyte( ). It blocks until all characters are available, or the end of line character is read. The read( ) function returns the number of characters read. The fd parameter is ignored.

`int` **`write`**`(int fd, char *buf, int nbytes)`

Writes nbytes bytes to the standard output by calling outbyte( ). It blocks until all characters have been written. The write( ) function returns the number of characters written. The fd parameter is ignored.

`int` **`isatty`**`(int fd)`

Reports if a file is connected to a tty. This function always returns 1, because only the stdin and stdout streams are supported.

`int` **`fcntl`** `(int fd, int cmd, long arg)`

A dummy implementation of fcntl, which always returns 0. fcntl is intended to manipulate file descriptors according to the command specified by cmd. Because Standalone does not provide a file system, this function is not used.

Cortex A9 gcc Errno Function

`int errno( )`

Returns the global value of errno as set by the last C library call.

Cortex A9 gcc Memory Management

`char *sbrk(int nbytes)`

Allocates nbytes of heap and returns a pointer to that piece of memory. This function is called from the memory allocation functions of the C library.

Cortex A9 gcc Process Functions

The functions getpid( ) in getpid.c and kill( ) in kill.c are included for completeness and because they are referenced by the C library.

Cortex A9 Processor-Specific Include Files

The xreg_cortexa9.h include file contains the register numbers and the register bits for the ARM Cortex-A9 processor.

The xpseudo_asm.h include file contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A9 Time Functions

The xtime_l.c file and corresponding xtime_l.h include file provide access to the 64-bit Global Counter in the PMU. This counter increases by one at every 2 processor cycles. The sleep.c file and corresponding sleep.h include file implement sleep functions. Sleep functions are implemented as busy loops.

Cortex A9 Time Function Summary

Cortex A9 Event Counters

xpm_counter.c and xpm_counter.h provide APIs for configuring and controlling the Cortex-A9

Performance Monitor Events. Cortex-A9 Performance Monitor has 6 event counters which can be used to count a variety of events described in Cortex-A9 TRM.

xpm_counter.h defines configurations (XPM_CNTRCFGx) which specifies the event counters to count a set of events.

Cortex A9 Event Counters Function Summary

The Event Counters functions are summarized below. Click on the function name to go to the description.

void Xpm_SetEvents(int PmcrCfg)

void Xpm_GetEventCounters(u32 *PmCtrValue)

Cortex A9 Event Counters Function Description

`void` **`Xpm_SetEvents`**`(int PmcrCfg)`

This function configures the Cortex A9 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

PmcrCfg is configuration value based on which the event counters are configured.

Use XPM_CNTRCFG* values defined in xpm_counter.h to define a configuration which specify the event counters to count a set of events.

`void` **`Xpm_GetEventCounters`**`(u32 *PmCtrValue)`

This function disables the event counters and returns the counter values.

PmCtrValue returns the counter values.

Xilinx Hardware Abstraction Layer

The following sections describe the Xilinx® Hardware Abstraction Layer API. It contains the following sections:

Types (xil_types)

Header File

```
#include "xil_types.h"
```

Typedef

```
typedef unsigned char u8
typedef unsigned short u16
typedef unsigned long u32
typedef unsigned long long u64
typedef char s8
typedef short s16
typedef long s32
typedef long long s64
```

Macros

Macro	Value
#define TRUE	1
#define FALSE	0
#define NULL	0
#define XIL_COMPONENT_IS_READY	0x11111111
#define XIL_COMPONENT_IS_STARTED	0x22222222

Register IO (xil_io)

Header File

```
#include "xil_io.h"
```

Common API

The following is a linked summary of register IO functions. They can run on MicroBlaze and Cortex A9 processors.

u8 Xil_In8(u32 Addr)
u16 Xil_EndianSwap16 (u16 Data)
u16 Xil_Htons(u16 Data)
u16 Xil_In16(u32 Addr)
u16 Xil_In16BE(u32 Addr)
u16 Xil_In16LE(u32 Addr)
u16 Xil_Ntohs(u16 Data)
u32 Xil_EndianSwap32 u32 Data)
u32 Xil_Htonl(u32 Data)
u32 Xil_In32(u32 Addr)
u32 Xil_In32BE(u32 Addr)
u32 Xil_In32LE(u32 Addr)
u32 Xil_Ntohs(u32 Data)
void Xil_Out8(u32 Addr, u8 Value)
void Xil_Out16(u32 Addr, u16 Value)
void Xil_Out16BE(u32 Addr, u16 Value)
void Xil_Out16LE(u32 Addr, u16 Value)
void Xil_Out32(u32 Addr, u32 Value)
void Xil_Out32BE(u32 Addr, u32 Value)
void Xil_Out32LE(u32 Addr, u32 Value)

`u8` **`Xil_In8`**`(u32 Addr)`

Perform an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address.

`u16` **`Xil_EndianSwap16`** `(u16 Data)`

Perform a 16-bit endian swapping.

Parameters:

Data contains the value to be swapped.

Returns:

Endian swapped value.

`u16` **`Xil_Htons`**`(u16 Data)`

Convert a 16-bit number from host byte order to network byte order.

Parameters:

Data the 16-bit number to be converted.

Returns:

The converted 16-bit number in network byte order.

`u16` **`Xil_In16`**`(u32 Addr)`

Perform an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address.

`u16` **`Xil_In16BE`**`(u32 Addr)`

Perform an big-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

`u16` **`Xil_In16LE`**`(u32 Addr)`

Perform a little-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is

`u16` **`Xil_Ntohs`**`(u16 Data)`

Convert a 16-bit number from network byte order to host byte order.

Parameters:

Data the 16-bit number to be converted.

Returns:

The converted 16-bit number in host byte order.

`u32` **`Xil_EndianSwap32`** `(u32 Data)`

Perform a 32-bit endian swapping.

Parameters:

Data contains the value to be swapped.

Returns:

Endian swapped value.

`u32` **`Xil_Htonl`**`(u32 Data)`

Convert a 32-bit number from host byte order to network byte order.

Parameters:

Data the 32-bit number to be converted.

Returns:

The converted 32-bit number in network byte order.

`u32` **`Xil_In32`**`(u32 Addr)`

Perform an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address.

`u32` **`Xil_In32BE`**`(u32 Addr)`

Perform a big-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

`u32` **`Xil_In32LE`**`(u32 Addr)`

Perform a little-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

`Addr` contains the address at which to perform the input operation.

Returns:

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

`u32` **`Xil_Ntohs`**`(u32 Data)`

Convert a 32-bit number from network byte order to host byte order.

Parameters:

Data the 32-bit number to be converted.

Returns:

The converted 32-bit number in host byte order.

`void` **`Xil_Out8`**`(u32 Addr, u8 Value)`

Perform an output operation for an 8-bit memory location by writing the specified value to the specified address.

Parameters:

`Addr` contains the address at which to perform the output operation.

`Value` contains the value to be output at the specified address.

`void` **`Xil_Out16`**`(u32 Addr, u16 Value)`

Perform an output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters:

`Addr` contains the address at which to perform the output operation.

`Value` contains the value to be output at the specified address.

`void` **`Xil_Out16BE`**`(u32 Addr, u16 Value)`

Perform a big-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters:

`Addr` contains the address at which to perform the output operation.

`Value` contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byte-swapped value is written to the address.

```
void
```

Exception (xil_exception)

Header File

```
#include "xil_exception.h"
```

Typedef

```
typedef void(* Xil_ExceptionHandler)(void *Data)
```

This typedef is the exception handler function pointer.

Common API

The following are exception functions. They can run on MicroBlaze and Cortex A9 processors.

`void` **`Xil_ExceptionDisable`**`()`

Disable Exceptions.

`void` **`Xil_ExceptionEnable`**`()`

Enable Exceptions.

`void` **`Xil_ExceptionInit`**`()`

Initialize exception handling for the processor. The exception vector table is set up with the stub handler for all exceptions.

`void` **`Xil_ExceptionRegisterHandler`**`(u32 Id, Xil_ExceptionHandler Handler, void *Data)`

Make the connection between the ID of the exception source and the associated handler that runs when the exception is recognized. `Data` is used as the argument when the handler is called.

Parameters:

`Id` contains the identifier (ID) of the exception source. This should be `XIL_EXCEPTION_INT` or be in the range of `0` to `XIL_EXCEPTION_LAST`. Refer to the `xil_exception.h` file for further information.

`Handler` is the handler for that exception.

`Data` is a reference to data that will be passed to the handler when it is called.

`void` **`Xil_ExceptionRemoveHandler`**`(u32 Id)`

Remove the handler for a specific exception ID. The stub handler is then registered for this exception ID.

Parameters:

`Id` contains the ID of the exception source. It should be `XIL_EXCEPTION_INT` or in the range of `0` to `XIL_EXCEPTION_LAST`. Refer to the `xil_exception.h` file for further information.

Common Macro

The common macro is:

```
#define XIL_EXCEPTION_ID_INT
```

This macro is defined for all processors and used to set the exception handler that corresponds to the interrupt controller handler. The value is processor-dependent. For example:

```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(XilExceptionHandler)IntcHandler, IntcData)
```

MicroBlaze Processor-Specific Macros

Macro	Value
#define XIL_EXCEPTION_ID_FIRST	0
#define XIL_EXCEPTION_ID_FSL	0
#define XIL_EXCEPTION_ID_UNALIGNED_ACCESS	1
#define XIL_EXCEPTION_ID_ILLEGAL_OPCODE	2
#define XIL_EXCEPTION_ID_IOPB_EXCEPTION	3
#define XIL_EXCEPTION_ID_IPLB_EXCEPTION	3
#define XIL_EXCEPTION_ID_DOPB_EXCEPTION	4
#define XIL_EXCEPTION_ID_DPLB_EXCEPTION	4
#define XIL_EXCEPTION_ID_DIV_BY_ZERO	5
#define XIL_EXCEPTION_ID_FPU	6
#define XIL_EXCEPTION_ID_MMU	7
#define XIL_EXCEPTION_ID_LAST	XIL_EXCEPTION_ID_MMU

Cache (xil_cache)

Header File

```
#include "xil_cache.h"
```

Common API

The functions listed in this sub-section can be executed on all processors.

`void` **`Xil_DCacheDisable`**`()`

Disable the data cache.

`void` **`Xil_DCacheEnable`**`()`

Enable the data cache.

`void` **`Xil_DCacheFlush`**`()`

Flush the entire data cache. If any cacheline is dirty (has been modified), it is written to system memory. The entire data cache will be invalidated.

`void` **`Xil_DCacheFlushRange`**`(u32 Addr, u32 Len)`

Flush the data cache for the given address range. If any memory in the address range (identified as `Addr`) has been modified (and are dirty), the modified cache memory will be written back to system memory. The cacheline will also be invalidated.

Parameters:

`Addr` is the starting address of the range to be flushed.

`Len` is the length, in bytes, to be flushed.

`void` **`Xil_DCacheInvalidate`**`()`

Invalidate the entire data cache. If any cacheline is dirty (has been modified), the modified contents are lost.

`void` **`Xil_DCacheInvalidateRange`**`(u32 Addr, u32 Len)`

Invalidate the data cache for the given address range. If the bytes specified by the address (Addr) are cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost.

Parameters:

`Addr` is address of range to be invalidated.

`Len` is the length in bytes to be invalidated.

`void` **`Xil_ICacheDisable`**`()`

Disable the instruction cache.

`void` **`Xil_ICacheEnable`**`()`

On MicroBlaze processors, enable the instruction cache.

`void` **`Xil_ICacheInvalidate`**`()`

Invalidate the entire instruction cache.

`void` **`Xil_ICacheInvalidateRange`**`(u32 Addr, u32 Len)`

Invalidate the instruction cache for the given address range.

Parameters:

`Addr` is address of range to be invalidated.

`Len` is the length in bytes to be invalidated.

Assert (xil_assert)

Header File

```
#include "xil_assert.h"
```

Typedef

```
typedef void(* Xil_AssertCallback)(char *Filename, int Line)
```

Common API

The functions listed in this sub-section can be executed on all processors.

`void` **`Xil_Assert`**`(char *File, int Line)`

Implement `assert`. Currently, it calls a user-defined callback function if one has been set. Then, potentially enters an infinite loop depending on the value of the `Xil_AssertWait` variable.

Parameters:

`File` is the name of the filename of the source.

`Line` is the line number within File.

`void` **`Xil_AssertSetCallbac`**`k(xil_AssertCallback Routine)`

Set up a callback function to be invoked when an assert occurs. If there was already a callback installed, then it is replaced.

Parameters:

`Routine` is the callback to be invoked when an assert is taken.

`#define` **`Xil_AssertVoid`**`(Expression)`

This assert macro is to be used for functions that do not return anything (void). This can be used in conjunction with the `Xil_AssertWait` boolean to accommodate tests so that asserts that fail still allow execution to continue.

Parameters:

`Expression` is the expression to evaluate. If it evaluates to false, the assert occurs.

`#define` **`Xil_AssertNonvoid`**`(Expression)`

This assert macro is to be used for functions that return a value. This can be used in conjunction with the `Xil_AssertWait` boolean to accommodate tests so that asserts that fail still allow execution to continue.

Parameters:

`Expression` is the expression to evaluate. If it evaluates to false, the assert occurs.

Returns:

Returns 0 unless the `Xil_AssertWait` variable is true, in which case no return is made and an infinite loop is entered.

#define **Xil_AssertVoidAlways**()

Always assert. This assert macro is to be used for functions that do not return anything (void). Use for instances where an assert should always occur.

Returns:

Returns void unless the `Xil_AssertWait` variable is true, in which case no return is made and an infinite loop is entered.

#define **Xil_AssertNonvoidAlways**()

Always assert. This assert macro is to be used for functions that return a value. Use for instances where an assert should always occur.

Returns:

Returns void unless the `xil_AssertWait` variable is true, in which case no return is made and an infinite loop is entered.

Extra Header File

The `xil_hal.h` header file is provided84 h1

XIL_TESTMEM_WALKONES

Walking Ones test.

This test uses a walking "1" as the test value for memory.

```
location 1 = 0x00000001
location 2 = 0x00000002
...
```

XIL_TESTMEM_WALKZEROS

Walking Zeros test.

This test uses the inverse value of the walking ones test as the test value for memory.

```
location 1 = 0xFFFFFFFE
location 2 = 0xFFFFFFFD
...
```

XIL_TESTMEM_INVERSEADDR

Inverse Address test.

This test uses the inverse of the address of the location under test as the test value for memory.

XIL_TESTMEM_FIXEDPATTERN

Fixed Pattern test.

This test uses the provided patterns as the test value for memory.

If zero is provided as the pattern, the test uses 0xDEADBEEF.

> ***Caution!*** The tests are DESTRUCTIVE. Run them before any initialized memory spaces have been set up. The address provided to the memory tests is not checked for validity, except for the case where the value is NULL. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.

Note: This test is used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 to the power of width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two, making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. If you need to test large blocks of memory, it is recommended that you break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Header File

```
#include "xil_testmem.h"
```

Common API

`int` **`Xil_Testmem32`** `(u32 *Addr, u32 Words, u32 pattern, u8 Subtest)`

Perform a destructive 32-bit wide memory test.

Parameters:

`Addr` is a pointer to the region of memory to be tested.

`Words` is the length of the block.

`Pattern` is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

`Subtest` is the test selected. See xil_testmem.h for possible values.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

Note: This test is used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 to the power of width, the patterns used in `XIL_TESTMEM_WALKONES` and `XIL_TESTMEM_WALKZEROS` repeat on a boundary of a power of two, making detecting addressing errors more difficult. This is true of the `XIL_TESTMEM_INCREMENT` and `XIL_TESTMEM_INVERSEADDR` tests also. If you need to test large blocks of memory, it is recommended that you break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Test Register IO (xil_testio)

This file contains utility functions to teach endian-related memory I/O functions.

Header File

```
#include "xil_testio.h"
```

Common API

`int` **`Xil_TestIO8`**`(u8 *Addr, int Len, u8 Value)`

Perform a destructive 8-bit wide register IO test where the register is accessed using `Xil_Out8` and `Xil_In8`. the `Xil_TestIO8` function compares the read and write values.

Parameters:

`Addr` is a pointer to the region of memory to be tested.

`Len` is the length of the block.

`Value` is the constant used for writing the memory.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

`int` **`Xil_TestIO16`**`(u8 *Addr, int Len, u16 Value, int Kind, int Swap)`

Perform a destructive 16-bit wide register IO test. Each location is tested by sequentially writing a 16-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence:

Xil_Out16LE/Xil_Out16BE, Xil_In16, Compare In-Out values, Xil_Out16, Xil_In16LE/Xil_In16BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Parameters:

`Addr` is a pointer to the region of memory to be tested.

`Len` is the length of the block.

`Value` is the constant used for writing the memory.

`Kind` is the test kind. Acceptable values are: `XIL_TESTIO_DEFAULT`, `XIL_TESTIO_LE`, `XIL_TESTIO_BE`.

`Swap` indicates whether to byte swap the read-in value.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

`int` **`Xil_TestIO32`**`(u8 *Addr, int Len, u32 Value, int Kind, int Swap)`

Perform a destructive 32-bit wide register IO test. Each location is tested by sequentially writing a 32-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence: Xil_Out32LE/Xil_Out32BE, Xil_In32, Compare In-Out values, Xil_Out32, Xil_In32LE/Xil_In32BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Parameters:

`Addr` is a pointer to the region of memory to be tested.

`Len` is the length of the block.

`Value` is the constant used for writing the memory.

`Kind` is the test kind. Acceptable values are: `XIL_TESTIO_DEFAULT`, `XIL_TESTIO_LE`, `XIL_TESTIO_BE`.

`Swap` indicates whether to byte swap the read-in value.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

Test Cache (xil_testcache)

This file contains utility functions to test the cache.

Header File

```
#include "xil_testcache.h"
```

Common API

`int` **`Xil_TestDCacheAll`**`()`

Tests the DCache related functions on all related API tests such as `Xil_DCacheFlush` and `Xil_DCacheInvalidate`. This test function writes a constant value to the data array, flushes the DCache, writes a new value, and then invalidates the DCache.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

`int` **`Xil_TestDCacheRange`**`()`

Tests the DCache range-related functions on a range of related API tests such as `Xil_DCacheFlushRange` and `Xil_DCacheInvalidate_range`. This test function writes a constant value to the data array, flushes the range, writes a new value, and then invalidates the corresponding range.

Returns:

0 is returned for a pass.

-1 is returned for a failure.

`int` **`Xil_TestICacheAll`**`()`

Perform `xil_icache_invalidate()`.

Returns:

0 is returned for a pass. The function will hang if it fails.

`int` **`Xil_TestICacheRange`**`()`

Perform `Xil_ICacheInvalidateRange()` on a few function pointers.

Returns:

0 is returned for a pass. The function will hang if it fails.

Hardware Abstraction Layer Migration Tips

Mapping Header Files to HAL Header Files

You should map the old header files to the new HAL header files as listed in Table 5.

Table 5: **HAL Header File Mapping**

Area	Old Header File	New Header File
Register IO	“xio.h“	“xil_io.h”
Exception	“xexception_l.h” “mb_interface.h”	“xil_exception.h”
Cache	“xcache.h” “mb_interface.h”	“xil_cache.h”
Interrupt	“xexception_l.h” “mb_interface.h”	“xil_exception.h”
Typedef u8 u16 u32	“xbasic_types.h”	“xil_types.h”
Typedef of Xuint32 Xfloat32 ...	“xbasic_types.h”	Deprecated. Do not use.
Assert	“xbasic_types.h”	“xil_assert.h”
Test Memory	“xutil.h”	“xil_testmem.h”
Test Register IO	None	“xil_testio.h”
Test Cache	None	“xil_testcache.h”
XTRUE, XFALSE, XNULL definitions	“xbasic_types.h”	Deprecated. Use TRUE, FALSE, NULL defined in xil_types.h

Mapping Functions to HAL Functions

You should map old functions to the new HAL functions as follows.

Table 6: **I/O Function Mapping**

Old xio	New xil_io
#include “xio.h”	#include “xil_io.h”
XIo_In8	Xil_In8
XIo_Out8	Xil_Out8
XIo_In16	Xil_In16
XIo_Out16	Xil_Out16
XIo_In32	Xil_In32
XIo_Out32	Xil_Out32

Table 7: **Exception Function Mapping**

Old xexception	New Xil_exception
#include “xexception_l.h” #include “mb_interface.h”	#include “xil_exception.h”
XExc_Init	Xil_ExceptionInit
XExc_mEnableException microblaze_enable_exceptions	For all processors: Xil_ExceptionEnable(void)
XExc_registerHandler microblaze_register_exception_handler	Xil_ExceptionRegisterHandler
XExc_RemoveHandler	Xil_ExceptionRemoveHandler
XExc_mDisableExceptions microblaze_disable_exceptions	Xil_ExceptionDisable

Table 8: **Interrupt Function Mapping**

Old Interrupt	New Xil_interrupt
#include “xexception_l.h”	#include “xil_exception.h”
XExc_RegisterHandler(XEXC_ID_NON_CRITICAL, handler) microblaze_register_handler(handler)	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, handler)

Table 9: **Cache Function Mapping**

Old xcache	New xil_cache
#include “Xcache_l.h” #include “mb_interface.h”	#include “xil_cache.h”
XCache_EnableDCache microblaze_enable_dcache	For all processors: Xil_DCacheEnable(void)
XCache_DisableDCache microblaze_disable_dcache	Xil_DCacheDisable
XCache_InvalidateDCacheRange microblaze_invalidate_dcache_range	Xil_DCacheInvalidateRange
microblaze_invalidate_dcache	Xil_DCacheInvalidate
XCache_FlushDCacheRange microblaze_flush_dcache_range	Xil_DCacheFlushRange
microblaze_flush_dcache	Xil_DCacheFlush
XCache_EnableICache microblaze_enable_icache	For all processors: Xil_ICacheEnable(void)
XCache_DisableICache microblaze_disable_icache	Xil_ICacheDisable
XCache_InvalidateICacheRange microblaze_invalidate_icache_Range	Xil_ICacheInvalidateRange
microblaze_invalidate_icache	Xil_ICacheInvalidate

Table 10: **Assert Function Mapping**

Old ASSERT	New xil_assert
#include “xbasic_types.h”	#include “xil_assert.h”
XAssert	Xil_Assert
XASSERT_VOID	Xil_AssertVoid
XASSERT_NONVOID	Xil_AssertNonvoid
XASSERT_VOID_ALWAYS	Xil_AssertVoidAlways
XASSERT_NONVOID_ALWAYS	Xil_AssertNonvoidAlways
XAssertSetCallback	Xil_AssertSetCallback

Table 11: **Memory Test Function Mapping**

Old XUtil_Memtest	New xil_testmem
#include “xutil.h”	#include “xil_testmem.h”
XUtil_MemoryTest32	Xil_Testmem32
XUtil_MemoryTest16	Xil_Testmem16
XUtil_MemoryTest8	Xil_Testmem8

Program Profiling

The Standalone OS supports program profiling in conjunction with the GNU compiler tools and the Xilinx Microprocessor Debugger (XMD). Profiling a program running on a hardware (board) provides insight into program execution and identifies where execution time is spent. The interaction of the program with memory and other peripherals can be more accurately captured.

Program running on hardware target is profiled using *software intrusive* method. In this method, the profiling software code is instrumented in the user program. The profiling software code is a part of the `libxil.a` library and is generated when software intrusive profiling is enabled in Standalone. For more details on about profiling, refer to the “Profile Overview” section of the *SDK Help.*

When the **`-pg`** profile option is specified to the compiler (`mb-gcc`), the profiling functions are linked with the application to profile automatically. The generated executable file contains code to generate profile information.

Upon program execution, this instrumented profiling function stores information on the hardware. XMD collects the profile information and generates the output file, which can be read by the GNU gprof tool. The program functionality remains unchanged but it slows down the execution.

Note: The profiling functions do not have any explicit application API. The library is linked due to profile calls (`_mcount`) introduced by GCC for profiling.

Profiling Requirements

- Software intrusive profiling requires memory for storing profile information. You can use any memory in the system for profiling.
- A timer is required for sampling instruction address. The `axi_timer` is the supported profile timer for MicroBlaze processors. The Global Timer is used as the profile timer for Cortex A9 processors.

Profiling Functions

`_profile_init`

Called before the application `main( )`

Configuring the Standalone OS

You can configure the Standalone OS using the Board Support Package Settings dialog box in SDK.

Table 12 lists the configurable parameters for the Standalone OS.

Table 12: **Configuration Parameters**

Parameter	Type	Default Value	Description
`enable_sw_intrusive_profiling`	Bool	false	Enable software intrusive profiling functionality. Select **`true`** to enable.
`profile_timer`	Peripheral Instance	none	Specify the timer to use for profiling. Select an **axi_timer** from the list of displayed instances. For Cortex A9, the ARM Cortex-A9 Private timer is used.
`stdin`	Peripheral Instance	none	Specify the **STDIN** peripheral from the drop down list
`stdout`	Peripheral Instance	none	Specify the **STDOUT** peripheral from the drop down list.
`predecode_fpu_exception`	Bool	false	This parameter is valid only for MicroBlaze processor when FPU exceptions are enabled in the hardware. Setting this to **`true`** will include extra code that decodes and stores the faulting FP instruction operands in global variables.

MicroBlaze MMU Example

The `tlb_add` function adds a single TLB entry. The parameters are:

tlbindex	The index of the TLB entry to be updated.
tlbhi	The value of the TLBHI field.
tlblo	The value of the TLBLO field.

```
static inline void tlb_add(int tlbindex, unsigned int tlbhi, unsigned int
tlblo)
{
    __asm__ __volatile__ ("mts rtlbx,  %2 \n\t"
                          "mts rtlbhi, %0 \n\t"
                          "mts rtlblo, %1 \n\t"
                          :: "r" (tlbhi),
                          "r" (tlblo),
                          "r" (tlbindex));

    tlbentry[tlbindex].tlbhi = tlbhi;
    tlbentry[tlbindex].tlblo = tlblo;
}
```

Given a base and high address, the tlb_add_entries function figures the minimum number page mappings/TLB entries required to cover the area. This function uses recursion to figure the entire range of mappings.

Parameters:

base	The base address of the region of memory
high	The high address of the region of memory
tlbaccess	The type of access required for this region of memory. It can be a logical or -ing of the following flags: 0 indicates read-only access TLB_ACCESS_EXECUTABLE means the region is executable TLB_ACCESS_WRITABLE means the region is writable

Returns: 1 on success and 0 on failure

```
static int tlb_add_entries (unsigned int base, unsigned int high, unsigned
int tlbaccess)
{
    int sizeindex, tlbsizemask;
    unsigned int tlbhi, tlblo;
    unsigned int area_base, area_high, area_size;
    static int tlbindex = 0;

    // Align base and high to 1KB boundaries
    base = base & 0xfffffc00;
   high = (high >= 0xfffffc00) ? 0xffffffff : ((high + 0x400) & 0xfffffc00)
- 1;

    // Start trying to allocate pages from 16 MB granularity down to 1 KB
    area_size   = 0x1000000;            // 16 MB
    tlbsizemask = 0x380;                // TLBHI[SIZE] = 7 (16 MB)

    for (sizeindex = 7; sizeindex >= 0; sizeindex--) {
        area_base = base & sizemask[sizeindex];
        area_high = area_base + (area_size - 1);

        // if (area_base <= (0xffffffff - (area_size - 1))) {

        if ((area_base >= base) && (area_high <= high)) {

            if (tlbindex < TLBSIZE) {
           tlbhi = (base & sizemask[sizeindex]) | tlbsizemask | 0x40;
// TLBHI: TAG, SIZE, V
            tlblo = (base & sizemask[sizeindex]) | tlbaccess | 0x8;
// TLBLO: RPN, EX, WR, W
                tlb_add (tlbindex, tlbhi, tlblo);

                tlbindex++;
            } else {
                // We only handle the 64 entry UTLB management for now
                return 0;
            }

            // Recursively add entries for lower area
            if (area_base > base)
                if (!tlb_add_entries (base, area_base - 1, tlbaccess))
                    return 0;
```

```
                // Recursively add entries for higher area
                if (area_high < high)
                    if (!tlb_add_entries(area_high + 1, high, tlbaccess))
                        return 0;

                break;
            }
            // else, we try the next lower page size
            area_size   = area_size >> 2;
            tlbsizemask = tlbsizemask - 0x80;
        }
        return 1;
    }
```

For a complete example, refer to:
`$XILINX_SDK/data/embeddedsw/lib/bsp/xilkernel_v6_1/src/src/arch/microblaze/mpu.c`.